Titre: Title:	Parallélisation sur processeur graphique des techniques d'adaptation de maillage basées sur la maximisation de la conformité à une métrique riemannienne
Auteur: Author:	William Bussière
Date:	2017
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bussière, W. (2017). Parallélisation sur processeur graphique des techniques d'adaptation de maillage basées sur la maximisation de la conformité à une métrique riemannienne [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2702/

UNIVERSITÉ DE MONTRÉAL

PARALLÉLISATION SUR PROCESSEUR GRAPHIQUE DES TECHNIQUES D'ADAPTATION DE MAILLAGE BASÉES SUR LA MAXIMISATION DE LA CONFORMITÉ À UNE MÉTRIQUE RIEMANNIENNE

WILLIAM BUSSIÈRE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

PARALLÉLISATION SUR PROCESSEUR GRAPHIQUE DES TECHNIQUES D'ADAPTATION DE MAILLAGE BASÉES SUR LA MAXIMISATION DE LA CONFORMITÉ À UNE MÉTRIQUE RIEMANNIENNE

présenté par : BUSSIÈRE William
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. BILODEAU Guillaume-Alexandre, Ph. D., président
M. GUIBAULT François, Ph. D., membre et directeur de recherche
M. DAGENAIS Michel, Ph. D., membre

DÉDICACE

À ma famille

REMERCIEMENTS

Tout d'abord, je remercie le Fond de Recherche du Québec – Nature et Technologies (FRQNT) et le Conseil de Recherches en Sciences Naturelles et en Génie du Canada (CRSNG) pour l'aide financière substantielle qu'ils m'ont apportée ces deux dernières années. Sans ces deux organismes, il n'aurait probablement pas été possible de choisir un projet de recherche qui rejoignait aussi bien mes centres d'intérêt.

Je remercie également les divers membres du laboratoire qui ont travaillé à me fournir des cas de test. Ils m'ont permis d'observer si la méthodologie développée dans cette recherche était valide dans la pratique. Ils m'ont également permis d'observer certaines limitations qui n'auraient pu être découvertes à l'aide des tests synthétiques. La génération de certains cas s'est révélée être plus difficile que prévu et, pourtant, ils ont toujours démontré la même inclination à vouloir m'aider, ce qui fût très apprécié.

Un merci tout particulier à Julien Dompierre qui a partagé beaucoup de son temps pour m'aider à parfaire mes connaissances du domaine. J'ai toujours senti le plaisir qu'il avait à m'enseigner l'adaptation de maillage comme s'il s'agissait d'un art. Ses conseils ont permis d'augmenter considérablement la qualité de la présente recherche.

Sans oublier mon directeur de recherche, François Guibault, qui a été d'une aide essentielle pour mener à bien mon projet de maîtrise. Ses innombrables révisions, conseils et propositions m'ont permis d'amener cette recherche au-delà de mes objectifs. Lorsque je rencontrais des difficultés, j'ai souvent été surpris par la simplicité et l'efficacité des pistes de solution qu'il avançait. Sa vision claire et sa grande expérience ont grandement été appréciées tout au long du projet.

RÉSUMÉ

En mécanique des fluides numérique, une manière d'accroitre la précision d'une solution approchée tout en évitant une consommation excessive de ressources informatiques consiste à adapter le maillage au comportement de la solution. Par un processus itératif de résolution-adaptation, la forme et la taille des éléments sont adaptées à la courbure de la solution approchée afin de réduire l'erreur d'interpolation. Avec la croissance en taille des problèmes étudiés en mécanique des fluides, des techniques de parallélisation ont dû être développées pour réduire le temps de calcul des solveurs. Toutefois, si les adaptateurs ne bénéficient pas d'un effort similaire de parallélisation, leur temps de calcul dominera le processus itératif de résolution-adaptation. L'objectif de la présente recherche est de déterminer si les GPU sont en mesure d'accélérer le processus d'adaptation tout en produisant des maillages d'une qualité équivalente aux méthodes actuellement exécutées par le CPU.

Par le lemme de Céa, on sait que l'erreur d'approximation est bornée par l'erreur d'interpolation. Produite par un estimateur de l'erreur *a posteriori*, la fonction métrique spécifie à l'adaptateur la longueur idéale d'une arête en un endroit donné pour une erreur d'interpolation donnée. Des opérations topologiques et géométriques permettent de modifier un maillage initial afin d'uniformiser la conformité de ses éléments à la métrique spécifiée. La présente recherche s'attardera particulièrement aux algorithmes de déplacement de nœuds qui démontrent déjà, en optimisation de maillage, une certaine facilité à être parallélisés sur le GPU. Toutefois, il est attendu que la manipulation des éléments non simpliciaux et que l'échantillonnage précis d'une fonction métrique 3D, caractéristiques nécessaires pour l'adaptation de maillages en mécanique des fluides numérique, causent certains problèmes sur ce type de processeur.

Un logiciel d'adaptation de maillage a été développé pour vérifier les hypothèses de cette recherche. Tout le processus d'adaptation est dirigé par la mesure de conformité à la métrique. Sur le CPU, la fonction métrique est échantillonnée par recherche locale dans un maillage de fond. On propose également une nouvelle technique d'échantillonnage pour la métrique sur le GPU sous forme de texture. Une série d'algorithmes de déplacement de nœuds sont implémentés et comparés sur les deux types de processeurs, dont le lissage laplacien, la descente du gradient et Nelder-Mead. Une nouvelle version du lissage laplacien pour l'adaptation de maillage dans une métrique riemannienne est proposée. Également, en plus de l'axe des nœuds, on propose deux nouveaux axes de parallélisation pour les algorithmes de déplacement de nœuds : l'axe des éléments et l'axe des positions. Les détails d'implémentation les

plus importants pour porter les algorithmes de déplacement de nœuds sur GPU sont abordés, tels que la traduction des structures de données et la synchronisation des processeurs. Deux technologies pour le développement de noyau de calcul seront comparées : GLSL et CUDA.

Les résultats montrent clairement que l'échantillonnage de la fonction métrique par recherche locale n'est pas praticable sur GPU à cause de son grand volume d'instructions et de la divergence qu'elle induit sur les fils d'exécution, contrairement à l'échantillonnage par texture qui démontre d'encourageants taux d'accélération. Bien que l'échantillonnage par texture présente de petits défauts de précision pour les fonctions métriques les plus difficiles, on démontre qu'en augmentant la résolution des textures dans une mesure raisonnable, il est possible de rejoindre la précision du maillage de fond. Quant aux algorithmes de déplacement de nœuds, les algorithmes d'optimisation locale se démarquent du lissage laplacien par leur efficacité, autant pour les maillages tétraédriques qu'hexaédriques. De plus, Nelder-Mead se démarque des algorithmes d'optimisation locale par sa rapidité sur le CPU et le GPU. Les nouveaux espaces de parallélisation explorés sur le GPU ont permis de doubler le taux d'accélération des algorithmes. Par exemple, la version multiaxe de la descente du gradient est 48 à 110 fois plus rapide que sa version séquentielle et la version multiélément de Nelder-Mead est environ 50 fois plus rapide que sa version séquentielle. En somme, le pipeline d'adaptation de maillage optimal retenu échantillonne la métrique dans une texture et déplace les nœuds à l'aide de la version multiélément de Nelder-Mead écrite en CUDA sur le GPU. Malgré la grande portabilité des implémentations GLSL face aux implémentations CUDA, elles présentent de moins bonnes accélérations pour le pipeline optimal et sont souvent instables.

ABSTRACT

In numerical fluid mechanics, a way to enhance the precision of an approximate solution while limiting the consumption of computer resources, is to adapt the mesh to the solution's behavior. By an iterative process of consecutive resolution-adaptation phases, the shape and size of the elements are adapted to the curvature of the approximate solution to reduce the interpolation error. With the ever increasing size of the problems studied in numerical fluid mechanics, multiple techniques of parallelization have been developed to reduce the time needed by solvers to compute a solution. However, if the adapters do not benefit from a similar parallelization effort, their computing time would dominate the overall resolution-adaptation iterative process. This study's objective is to determine if GPUs are able to accelerate the adaptation process while producing meshes whose quality matches the one produced by current methods on the CPU.

By Céa's lemma, it is known that the approximation error is bounded by the interpolation error. Generated by an *a posteriori* error estimator, the metric function specifies to the adapter the ideal length of an edge in a given location to obtain a specific interpolation error. Topological and geometrical operations modify an initial mesh in order to uniformize the conformity of its elements to the specified metric. The present study will specifically focus on node relocation algorithms that already demonstrate, in the field of mesh optimization, an ease to be parallelized on GPUs. Although, it is predicted that the manipulation of non simplicial elements and that the precise sampling of a 3D metric function, which are necessary to the adaptation of meshes in numerical fluid mechanics, may cause certain problems for that type of processor.

A mesh adaptation software was developed to verify the hypotheses of this research. The entire adaptation process is driven by the measure of metric conformity. On the CPU, the metric function is sampled in a background mesh by local search. The research also proposes to sample the metric function in a texture on the GPU side. A series of node relocation algorithms are implemented and compared on both types of processor, including the Laplacian smoothing, the gradient descent and Nelder-Mead. A new version of the Laplacian smoothing for mesh adaptation in a Riemannian metric is proposed. Furthermore, in addition to the node axis, this study proposes two new parallelization axes for node relocation algorithms: the elements and positions axes. The most important implementation details to port the node relocation algorithms on the GPU are discussed, including processors synchronization and the translation of data structures. Two technologies for the development

of GPU kernels are compared: GLSL and CUDA.

The results clearly show that the sampling of the metric function by local search is not practical on the GPU because the method is too heavy in terms of instruction count and thread divergence. On the contrary, sampling this function through a texture demonstrates encouraging rates of acceleration. Even though texture sampling presents some precision flaws for the most difficult metric functions, this study demonstrates that by increasing the resolution of the textures by a reasonable amount, it is possible to meet the background mesh's precision. Concerning the node relocation algorithms, the local optimization algorithms stand out from Laplacian smoothing algorithms by their efficacy, as much on tetrahedral than hexahedral meshes. Furthermore, Nelder-Mead stands out from the local optimization algorithms by its high speed on the CPU as well as on the GPU. The newly explored parallelization axes on the GPU has doubled implementations acceleration rates. For example, the gradient descent's multiaxis version is between 48 and 110 times faster than its sequential version while Nelder-Mead's multielement version is approximately 50 times faster than its sequential version. In short, the selected optimal adaptation pipeline samples the metric function in a texture. It uses Nelder-Mead's multielement version, written in CUDA, to move mesh nodes on the GPU. Despite GLSL's high portability compared to CUDA, its implementation of the optimal pipeline shows lower acceleration rates and is often unstable.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface
CAO	Conception Assistée par Ordinateur
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GETMe	Geometric Element Transformation Method
GLSL	OpenGL Shader Language
GPU	Graphics Processing Unit
GPGPU	General-Purpose Processing on Graphics Processing Units
HLSL	High Level Shader Language
MEF	Méthode des Éléments Finis
MDF	Méthode des Différences Finies
MIMD	Multiple Instructions Multiple Data
MVF	Méthode des Volumes Finis
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

1.1 Contexte du projet

La mécanique des fluides numérique permet d'étudier une multitude de situations où le mouvement d'un fluide tel que l'air ou l'eau est le phénomène d'intérêt. Cette méthode d'analyse est notamment utilisée en ingénierie pour étudier l'écoulement d'air autour d'une aile d'avion ou le passage de l'eau dans une turbine hydroélectrique. Elle apporte également une contribution substantielle à rendre les jeux vidéos et les films d'animation plus réalistes, que ce soit en simulant l'évolution des nuages dans le ciel ou les tumultes d'une rivière. Les premières étapes pour préparer ce type de simulation sont la modélisation suivie de la discrétisation de la géométrie du problème. La discrétisation peut se faire à l'aide de maillages structurés, non structurés, hybrides ou plus marginalement à l'aide d'ondelettes. Puisque les problèmes en mécanique des fluides numérique sont généralement résolus à l'aide de la Méthode des Volumes Finis (MVF), la Méthode des Éléments Finis (MEF) ou la Méthode des Différences Finies (MDF) et puisqu'on veut modéliser des géométries arbitrairement complexes, la présente recherche s'attardera exclusivement aux maillages tridimensionnels non structurés. Ces maillages divisent l'espace de calculs en éléments simples comme des tétraèdres, des pyramides, des prismes et des hexaèdres. Le problème, complexe dans sa globalité, est résolu à l'aide de petits calculs simples sur chacun des éléments du maillage.

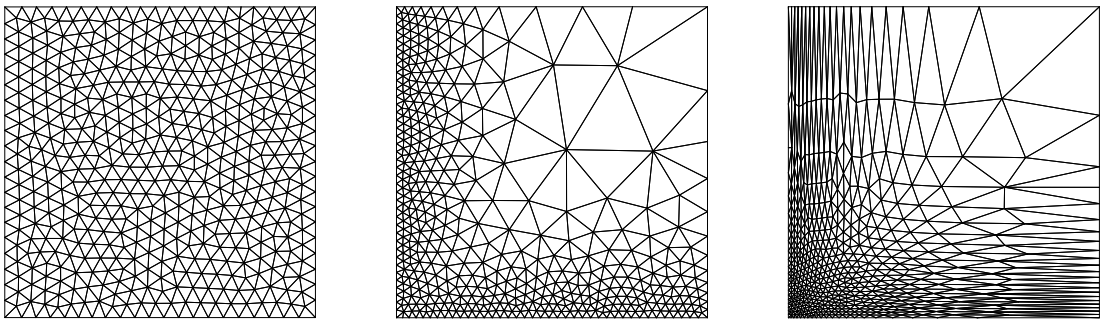
L'obtention d'une « bonne » discrétisation de la géométrie ne passe pas seulement par le type de modèle utilisé, mais aussi par la forme et la taille des éléments qui le constituent. La mécanique des fluides numérique vise à analyser un phénomène continu à l'aide d'une représentation discrète de l'espace. À l'exception de quelques cas triviaux, la solution obtenue sera toujours entachée d'erreurs d'approximation. Plus la taille des éléments diminue, plus l'erreur diminue, mais plus il faudra un grand nombre d'éléments pour remplir le même espace. Le temps de calcul et la consommation en mémoire des solveurs sont directement liés à la taille des maillages, c.-à-d. au nombre d'éléments qu'ils contiennent. On se retrouve alors à devoir faire un compromis entre la précision de notre solution et la consommation en ressources de notre méthode de résolution.

C'est pourquoi plusieurs techniques d'adaptation de maillage ont été développées. Leur but est de déterminer d'abord quelles seraient la taille et la forme de l'élément idéal qui couvre une portion donnée de l'espace, puis de construire un maillage qui respecte ces prescriptions de tailles et de formes sur l'ensemble de l'espace. On appelle *fonction métrique* cette carte de prescriptions, qui est en pratique stockée dans une structure de données tridimensionnelle.

Si l'on utilise des éléments linéaires pour interpoler la solution, l'erreur d'interpolation est guidée par la «courbure» de la solution. Pour minimiser l'erreur d'interpolation, on choisira d'utiliser de petits éléments là où la dérivée seconde est grande. Inversement, pour minimiser la consommation de ressources, on choisira d'utiliser de grands éléments là où la dérivée seconde est petite. La figure 1.1 illustre trois maillages 2D générés pour résoudre le même problème de réaction-diffusion. Seuls les maillages (b) isotrope et (c) anisotrope sont dits adaptés. L'anisotropie d'un maillage désigne le fait que la métrique en un point de l'espace prescrit des dimensions différentes selon la direction observée, ce qui donne des éléments plus ou moins étirés.

On veut adapter le maillage à la courbure de la solution, mais bien sûr on ne connaît pas cette solution a priori. Il est toutefois possible de procéder de manière itérative en deux phases : résolution et adaptation. On résout le problème d'abord sur un maillage grossier non adapté. Selon la courbure de la solution approchée, on construit une première métrique et l'on adapte le maillage à celle-ci. Ce maillage adapté permet d'obtenir une meilleure approximation de la solution ainsi qu'une meilleure approximation de sa courbure. Cela permettra d'adapter le maillage davantage, d'obtenir encore une fois une meilleure approximation de la solution et ainsi de suite jusqu'à ce que la solution soit suffisamment précise. Les deux phases sont parfois exécutées par deux logiciels différents. Le solveur résout le problème aux dérivées partielles à l'aide de la MVF, la MEF ou la MDF tandis que l'adaptateur s'occupe de modifier le maillage pour lui permettre de mieux capturer le phénomène à l'étude. Le cycle complet d'adaptation est illustré à la figure 1.2 avec les logiciels impliqués à chacune des phases.

Bien que le processus soit efficace pour résoudre des problèmes en mécanique des fluides avec une grande précision, converger à la précision voulue peut prendre beaucoup de temps.



(a) Uniforme (b) Isotrope (c) Anisotrope

Figure 1.1 Maillages d'un domaine carré avec différents degrés d'adaptation [1]

1. Tirées de Labbé et al. (2011) avec la permission écrite des auteurs

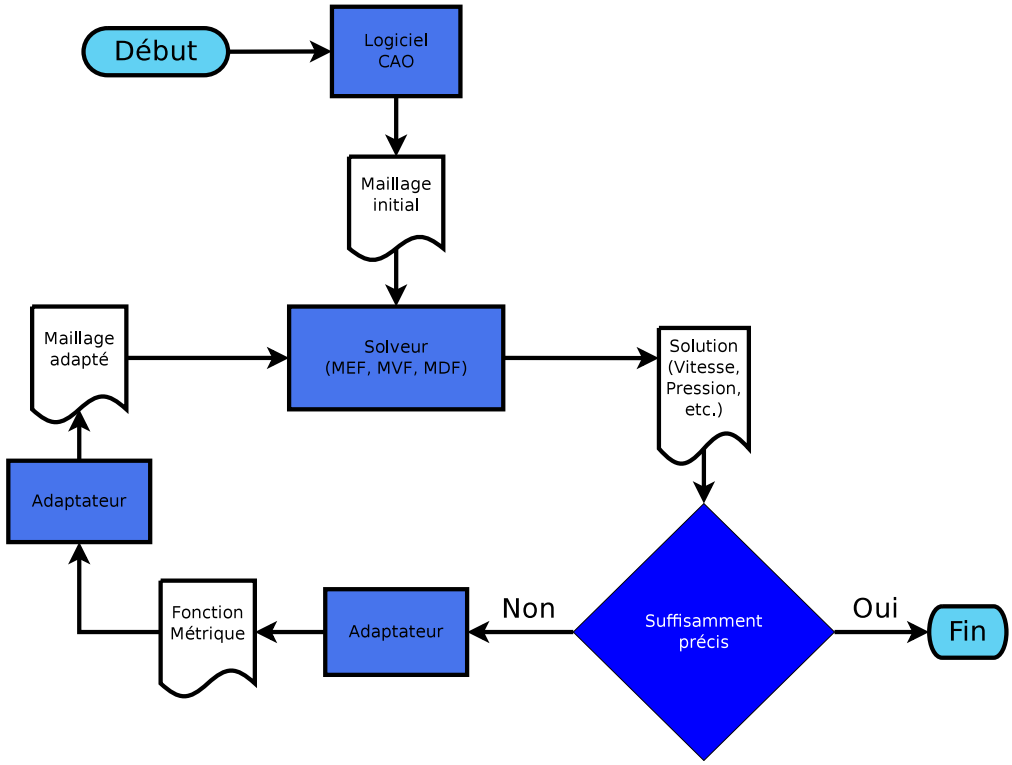


Figure 1.2 Cycle de résolution-adaptation pour réduire l'erreur d'interpolation

Concrètement, sur des maillages de plusieurs millions de nœuds, on parle de l'ordre de quelques heures pour un problème stationnaire et jusqu'à plusieurs jours pour les cas instationnaires. C'est pourquoi, suivant l'évolution des méthodes de résolution et la croissance des problèmes en taille, un effort considérable a été investi dans la parallélisation des solveurs. Notamment, il est maintenant possible de paralléliser les calculs en utilisant plusieurs fils d'exécution sur un ordinateur multicœur, plusieurs processus sur un superordinateur et des noyaux de calcul sur un Graphics Processing Unit (GPU). Le fait qu'un maillage soit constitué d'une multitude d'éléments rend sa manipulation particulièrement bien adaptée au traitement parallèle. Sur les ordinateurs multicœurs et les systèmes distribués, on va partitionner le maillage en un petit nombre de régions de tailles similaires (voir figure 1.3), traiter simultanément ces régions et finalement recoller le tout en propageant les données à leurs interfaces. Vu son architecture matérielle tout à fait unique, le GPU demande quant à lui un partitionnement bien différent. Dans son cas, il est préférable de partitionner le maillage en un grand nombre de petites régions afin de fournir du travail à ses centaines, voir ses milliers de cœurs.

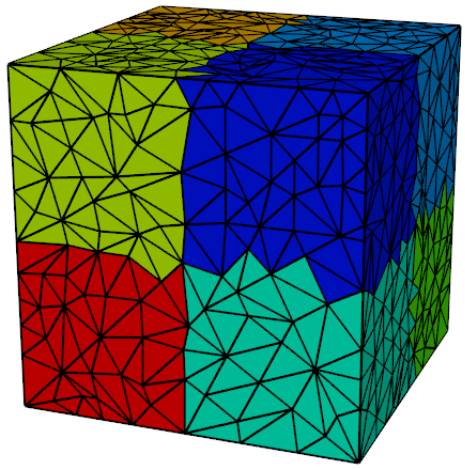


Figure 1.3 Partitionnement d'un maillage pour une résolution parallèle

1.2 Éléments de la problématique

Lorsque l'on simule un problème stationnaire sur un seul processeur, le temps nécessaire pour la résolution est significativement plus long que le temps requis pour l'adaptation. Puisqu'adapter le maillage n'est pas essentiel pour résoudre un problème en mécanique des fluides numérique, la majeure partie de l'effort de parallélisation a été concentrée sur les solveurs. Malheureusement, si un effort comparable n'est pas investi dans la parallélisation des méthodes d'adaptation de maillage, cette étape déterminera fortement le temps total du processus itératif d'adaptation-résolution. De plus, la résolution des équations aux dérivées partielles et l'adaptation de maillage sont deux problèmes de natures très différentes. Il n'est pas possible d'appliquer à l'un les techniques de parallélisation développées pour l'autre.

L'adaptation de maillage, qui consiste à optimiser un maillage selon des critères de forme et de taille locaux, comporte deux types d'opérations : les opérations topologiques et les opérations géométriques. Les opérations topologiques visent à régulariser la connectivité des nœuds. On peut insérer des nœuds, supprimer des nœuds ou échanger un petit groupe d'arêtes pour un autre. Cependant, il est important de conserver la conformité du maillage à travers ces opérations. Un maillage est conforme si ses éléments couvrent tout le domaine borné par le modèle géométrique et si l'intersection de deux éléments quelconques est soit vide ou composée au plus de leurs arêtes en 2D ou de leurs faces en 3D. Donc, les éléments doivent couvrir l'intégralité du domaine tout en évitant de se superposer. C'est une condition nécessaire pour assurer la validité des solutions obtenues par la MVF ou la MEF. Les opérations

géométriques visent quant à elles à régulariser la forme des éléments simplement en relocalisant leurs sommets. Elles préservent toujours le nombre de nœuds dans le maillage et leur connectivité. Il existe une variété d'algorithmes de déplacement de nœuds. Ils sont soit basés sur des heuristiques de déplacement soit sur des approches d'optimisation locale. On désignera également les algorithmes de déplacement de nœuds dans ce document par l'expression « lissage » (« mesh smoothing » en anglais).

Freitag et al. (1999) ont ouvert la voie au développement d'algorithmes parallèles pour l'optimisation et l'adaptation de maillages en énonçant une simple règle de base : un élément ne doit jamais voir plus d'un de ses nœuds être déplacé simultanément. La violation de cette règle entraîne la possibilité de produire involontairement des éléments inversés. La présence d'éléments inversés implique nécessairement la non-conformité d'un maillage, celui-ci devenant alors inutilisable pour les solveurs. Depuis la publication de cet article, bon nombre de recherches ont été publiées sur la parallélisation en 2D et 3D d'algorithmes de déplacement de nœuds sur des ordinateurs multicœurs et des systèmes distribués. Toutefois, ce n'est que récemment que l'on a observé un virage vers des plateformes moins conventionnelles comme le GPU.

L'évolution des GPU au fil des décennies n'a pas été linéaire. Dans les années 70, il s'agissait essentiellement de contrôleurs pour périphériques d'affichage. Puis, au cours des années 80, ils ont commencé à pouvoir traiter des instructions d'affichage et générer des lignes, des arcs et des rectangles. Les années 90 ont vu naître les premières Application Programming Interface (API) standardisées pour la génération de graphiques 3D telles qu'OpenGL et Direct3D. C'est dans les années 2000 que les GPU sont devenus des processeurs massivement parallèles hébergeant des dizaines et même des centaines de cœurs. Les scientifiques ont compris l'intérêt qu'ils représentaient pour le calcul de haute performance, mais le seul moyen d'en tirer avantage à l'époque était d'exprimer les problèmes sous forme de primitives géométriques et d'utiliser les nuanceurs de pixel (pixel shaders) pour effectuer les calculs. Ce sont les langages de programmation de haut niveau pour GPU qui ont réellement ouvert la voie au calcul généralisé sur processeurs graphiques (General-Purpose Processing on Graphics Processing Units (GPGPU)). La venue du langage CUDA en 2006, puis d'OpenCL en 2008 et finalement des nuanceurs de calcul (compute shaders) dans les API graphiques OpenGL et Direct3D ont démocratisé leur utilisation. Les GPU ont désormais des utilités diverses qui voient leur unification dans la programmabilité toujours croissante de leur processeur. De nos jours, pratiquement tous les ordinateurs, du cellulaire au superordinateur en passant par l'ordinateur de bureau, possèdent un ou plusieurs GPU. L'omniprésence ainsi que la flexibilité croissante des GPU expliquent l'engouement que ces processeurs suscitent en calcul de haute performance ces dernières années.

Pour souligner les différences architecturales entre les Central Processing Unit (CPU) et les GPU quant à l'organisation de leurs unités de traitement, on dira des premiers qu'ils sont *multi-cores* et des seconds qu'ils sont *many-cores*. Les GPU fonctionnent essentiellement en mode Single Instruction Multiple Data (SIMD), ou Single Instruction Multiple Threads (SIMT) selon la terminologie de *NVIDIA*. Mais contrairement aux CPU, il existe une hiérarchie stricte entre les cœurs d'un GPU. Ils sont regroupés en *Streaming Multiprocessors* qui contiennent chacun plusieurs *Streaming Processors* qui exécutent chacun plusieurs fils d'exécution simultanément. Un *warp* (groupe indivisible de 32 fils d'exécution) ne possède qu'un seul pointeur d'instruction et tous les fils d'exécution dont il a la responsabilité en dépendent. Lorsque les fils d'exécution d'un même *warp* évaluent une condition de branchement différemment, par exemple à la rencontre d'un if, $ifelse$, $while$, for ou d'un $switch$, le *warp* doit parcourir toutes les branches l'une après l'autre. L'expression consacrée en GPGPU pour parler de ce phénomène est la divergence. Il va de soi qu'un programme dont les exécutions divergent fréquemment observera une perte de parallélisme et par conséquent une perte d'efficience. Les CPU multicœurs, puisqu'ils fonctionnent surtout en mode Multiple Instructions Multiple Data (MIMD), ne connaissent pas cette inefficacité liée à l'exécution de branches étant donné que leurs cœurs sont autonomes.

La nature massivement parallèle et l'omniprésence des GPU rendent leur utilisation très attrayante pour la parallélisation des algorithmes d'adaptation de maillage par déplacement de nœuds. Quelques travaux ont déjà été réalisés dans ce sens, sans toutefois répondre pleinement aux besoins de l'industrie. D'Amato and Vénere (2013), Mei et al. (2014), Dahal and Newman (2014) et Cheng et al. (2015) ont démontré qu'il était praticable de porter les algorithmes de déplacement de nœuds sur le GPU. Malheureusement, leurs algorithmes ne prennent en compte aucune métrique et présentent alors peu d'utilités en mécanique des fluides. Jusqu'ici, seuls Rokos et al. (2011) paraissent avoir intégré une métrique. Leur implémentation ne supporte que les triangulations 2D pour l'instant. Ils ont laissé en travail futur la gestion des maillages 3D.

Le problème d'adaptation de maillages tridimensionnels multiéléments (formés de tétraèdres, de prismes, de pyramides et d'hexaèdres) est théoriquement bien posé pour être parallélisé sur GPU. Un coloriage de graphe permet de déterminer quels sont les nœuds qui peuvent être déplacés simultanément. Hormis les nœuds frontières, tous les nœuds du maillage qui sont traités en parallèle connaissent une exécution plus ou moins uniforme, ce qui est essentiel pour obtenir de bonnes performances sur GPU. Les difficultés surviennent lorsqu'on tente de tenir compte de la fonction métrique. Pour évaluer la métrique à un endroit donné, on doit trouver l'élément du maillage de fond qui contient l'échantillon et interpoler la métrique sur cet élément. Non seulement cette opération engendre une divergence des fils d'exécution, mais

il s'agit de l'opération la plus invoquée de tout le processus d'adaptation. Une transposition naïve vers le GPU des algorithmes d'adaptation de maillage anisotrope par déplacement de nœuds, à l'origine développés pour le CPU, offre des performances désastreuses.

Il est impératif d'étudier de nouvelles techniques d'évaluation de la métrique mieux adaptées au GPU si l'on veut pouvoir bénéficier de toute la puissance de calcul qu'ils offrent, soit un peu plus de 10 TFLOPS en simple précision pour une carte graphique TITAN X de *NVIDIA*. Rokos et al. (2011) constatant le même problème, ils proposent de stocker la métrique sous forme de texture avant d'échantillonner et d'interpoler celle-ci par le biais d'unités matérielles dédiées sur le GPU. Cependant, les auteurs ne semblent pas avoir analysé l'impact de cette nouvelle représentation sur la qualité des maillages adaptés. Il reste donc plusieurs études à faire pour déterminer la meilleure manière d'échantillonner la métrique sur GPU.

1.3 Hypothèses de recherche

On se concentrera dans cette recherche spécifiquement sur les algorithmes d'adaptation de maillage par déplacement de nœuds. Bien que ces algorithmes s'inscrivent dans un processus plus général d'adaptation qui comprend les opérations topologiques, celles-ci ne seront pas étudiées puisqu'elles sont difficilement parallélisables sur GPU.

La première hypothèse concerne la performance de la solution. Pour un algorithme de déplacement de nœuds donné, il doit être possible de produire une implémentation GPU fournissant une plus grande accélération qu'une implémentation *multithread* sur CPU. Des processeurs de même génération et de gamme similaires seront utilisés pour la comparaison.

La deuxième hypothèse s'attarde quant à elle à la qualité des maillages produits sur le GPU. Il n'y a pas lieu de croire que cette plateforme soit en mesure de produire des maillages de meilleure qualité. Toutefois, il est désirable que les maillages soient de qualité comparable. Si ce n'est pas le cas, il faut au moins que l'implémentation reste convergente, mathématiquement parlant, et qu'une allocation plus grande de ressources permette d'atteindre éventuellement les mêmes niveaux de qualité qu'une implémentation traditionnelle.

Étant donné que le déplacement de nœuds fait partie d'un processus plus large d'adaptation de maillage, le coût d'intégration de la solution doit être négligeable. C'est-à-dire que l'introduction de la solution dans un cadre fonctionnel et optimisé d'adaptation de maillage ne doit pas introduire de latence suffisamment grande ou requérir des calculs de conversion suffisamment coûteux pour annuler le gain en temps de calcul offert par la solution.

1.4 Objectifs de recherche

Un logiciel d'adaptation de maillage par métrique riemannienne sera développé. Le logiciel sera écrit en C++ puisque ce langage de haut niveau offre un grand contrôle sur la performance et donne directement accès au plus large éventail de technologies GPU. Pour chaque algorithme, des implémentations en Compute Unified Device Architecture (CUDA) et en OpenGL Shader Language (GLSL) seront développées pour déterminer si les technologies GPU sont équivalentes entre elles.

Les algorithmes développés devront correspondre à l'état de l'art actuel en adaptation de maillage par métrique riemannienne. Chacun de ces algorithmes se déclinera en quatre implémentations : séquentielle pour CPU, parallèle pour CPU, GLSL pour GPU et CUDA pour GPU. On s'assurera que les différentes implémentations d'un même algorithme aient des efficacités équivalentes avant de comparer leurs rapidités.

Afin de déterminer si la solution GPU proposée n'affecte pas significativement la capacité de l'adaptateur à produire des maillages de bonne qualité, il faudra définir une base de comparaison mathématiquement rigoureuse. Cette méthode de vérification indiquera l'ampleur de l'erreur en fonction de la difficulté de la fonction métrique à respecter.

Le logiciel développé doit permettre d'intégrer une implémentation GPU du déplacement de nœuds sans introduire de coûts supplémentaires considérables. C'est pourquoi des opérations topologiques classiques seront développées dans le logiciel. On testera différents agencements d'opérations topologiques et géométriques. Les opérations topologiques serviront également à préparer les cas de test de l'analyse des résultats.

Puisque le logiciel d'adaptation est un produit central de la présente recherche, sont code source est ouvert et publié sur le site internet GitHub : `https://github.com/wibus/GpuMesh`. Une courte description des classes importantes est fournie à l'annexe A pour faire le lien entre les concepts présentés dans ce mémoire et leur implémentation dans le logiciel.

1.5 Plan du mémoire

Le chapitre 2 du présent mémoire est une revue critique de la littérature entourant l'adaptation de maillage. On abordera d'abord les concepts mathématiques qui justifient les bénéfices promis par l'adaptation de maillage. Puis, on adoptera une perspective plus pratique du processus en présentant les mesures de la qualité, les opérations topologiques et les opérations géométriques. On poursuivra par une revue des efforts de parallélisation réalisés jusqu'à ce jour en adaptation de maillage. On évaluera finalement le niveau d'atteinte des objectifs de

recherche des différentes implémentations d'adaptation de maillage sur GPU disponibles à ce jour.

Le chapitre 3 décrit l'infrastructure logicielle développée dans le cadre de cette recherche. On y décrit les modules de l'adaptateur, chacun encapsulant une opération du processus d'adaptation, avant de présenter leurs instances respectives. On discutera aussi brièvement de l'ordonnancement optimal des opérations d'adaptation.

Le chapitre 4 complète la description de l'infrastructure logicielle par la présentation des modules GPU. On y parle des différences entre les modèles d'exécution du CPU et du GPU et de la manière dont cela influence la traduction des structures de données et des algorithmes. On parle également des difficultés qui ont fait surface pendant l'implémentation des algorithmes GPU, de leurs conséquences sur l'atteinte des objectifs de recherche et des solutions qui se sont imposées.

Le chapitre 5 présente les résultats comparatifs entre les différentes implémentations séquentielles et parallèles développées pour le CPU et le GPU. Une attention particulière est portée non seulement aux performances des algorithmes, mais aussi à leur efficacité à produire des maillages de bonne qualité.

Le dernier chapitre fait un retour sur la méthode développée, discute de son efficacité et rappelle les difficultés posées par l'introduction d'une métrique riemannienne dans le processus d'adaptation de maillage sur GPU. Des travaux qui mèneraient à un meilleur agencement du processus d'adaptation au GPU, tout en garantissant son intégrité, sont proposés.

CHAPITRE 2 ADAPTATION DE MAILLAGE

Puisqu'en mécanique des fluides numérique, il n'est généralement pas possible d'expliciter la solution des équations de Navier-Stokes, on construit des problèmes approchés en discrétisant le problème de base et en calculant des approximations sur ceux-ci à l'aide de la méthode de résolution de notre choix. Toute solution obtenue par la MVF, la MEF ou la MDF, les méthodes les plus populaires en mécanique des fluides numérique, est entachée d'erreur. Cette erreur provient principalement de deux sources : la discrétisation de l'espace en éléments de taille finie et l'utilisation de fonctions de base de degré fini. Il faut toutefois reconnaître que le choix du modèle physique peut avoir un impact important selon le phénomène étudié. Bien qu'il soit impossible de représenter exactement la solution, il est possible d'estimer l'erreur commise par la solution approchée et de la réduire jusqu'à l'amener sous un seuil prescrit. Cette tâche est déléguée au processus d'adaptation de maillage.

Ce chapitre explique comment l'erreur de discrétisation est estimée puis réduite à l'aide d'opérations topologiques et géométriques. La dernière section du chapitre passe en revue les différentes approches mises au point jusqu'à ce jour pour paralléliser les algorithmes de déplacement de nœuds. De plus, on y retrouve une description de l'architecture du GPU et les concepts informatiques nécessaires pour programmer ce type de processeur. Cela nous permettra de comprendre l'état actuel de la recherche en parallélisation d'algorithmes de déplacement de nœuds sur GPU.

2.1 Contrôle de l'erreur et lemme de Céa

Le lemme de Céa garantit une borne supérieure sur l'erreur d'approximation commise par notre solution approchée au problème d'équations aux dérivées partielles. Autrement dit, ce lemme nous assure que la solution obtenue par la MEF est une bonne représentation du phénomène étudié, sous certaines conditions.

Établissons ici le cadre mathématique de notre problème. On cherche à déterminer la fonction solution u telle que :

$$\begin{cases} Au = f \text{ sur } \Omega \\ \partial u/\partial n = h \text{ sur } \Gamma_h \\ u = g \text{ sur } \Gamma_g \end{cases} \tag{2.1}$$

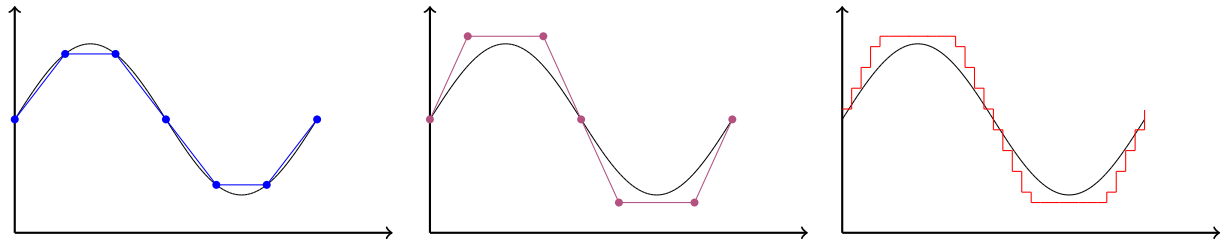
où $\Gamma_h \cap \Gamma_g = \emptyset$, $\Gamma_h \cup \Gamma_g = \Gamma$ et Γ est la frontière de l'ouvert Ω. Le lemme de Céa ne s'applique que si le problème aux limites possède une solution unique. Ainsi, pour que la démarche reste valide, il faut démontrer l'unicité des solutions approchées et démontrer que celles-ci convergent vers u si l'on utilise des éléments de plus en plus petits. Dans la pratique, ce sont les solveurs qui calculent les solutions approchées tandis que les adaptateurs génèrent des problèmes approchés qui représentent de mieux en mieux le problème d'origine.

Les problèmes approchés sont des discrétisations du problème d'origine. Le domaine est d'abord divisé en éléments simples (le maillage) puis la solution est interpolée sur ces éléments à l'aide de fonctions de base. Cette procédure introduit plusieurs types d'erreurs dans la recherche de u. D'abord, l'erreur d'*interpolation* désigne l'incertitude d'une valeur interpolée entre deux valeurs exactes. Dans notre cas, ce sont les fonctions d'interpolation utilisées sur les éléments qui ne représentent pas exactement u. L'erreur d'*approximation* comprend l'erreur d'interpolation, mais ajoute de l'incertitude aux nœuds du maillage. Finalement, l'erreur de *discrétisation* regroupe les erreurs de quantification liées à la représentation des nombres à virgule sur un ordinateur, soit l'erreur de troncature et l'erreur d'arrondi. La figure 2.1 illustre les différents types d'erreurs rencontrés en analyse numérique.

On peut calculer l'erreur d'approximation ϵ, en fonction de la solution réelle u et de la solution approchée u_h sur le domaine Ω, par la norme suivante :

$$\epsilon = \|u - u_h\|_{\Omega, L_\infty}, \text{ avec } u_h \in V_h \tag{2.2}$$

où V_h est l'espace des solutions approchées et L_∞ est l'espace donnant la distance maximale entre les solutions réelle et approchée. Or, le lemme de Céa nous assure que cette erreur est inférieure ou égale à la distance entre u et sa projection sur V_h, soit $\Pi_h u$, à une constante près (Cea (1964)) :



(a) Erreur d'interpolation (b) Erreur d'approximation (c) Erreur de discrétisation

Figure 2.1 Les trois principaux types d'erreurs rencontrer en analyse numérique

$$\|u - u_h\|_{\Omega,L_\infty} \leq C\|u - \Pi_h u\|_{\Omega,L_\infty} \tag{2.3}$$

De plus, on sait que la distance entre $\Pi_h u$ et u est inférieure ou égale à l'erreur d'interpolation, donnée par la distance entre u et $I_h u$:

$$\|u - \Pi_h u\|_{\Omega,L_\infty} \leq \|u - I_h u\|_{\Omega,L_\infty} \tag{2.4}$$

Donc, l'erreur d'approximation est bornée supérieurement par l'erreur d'interpolation à une constante près. En somme, pour minimiser l'erreur d'approximation, il faut d'abord minimiser l'erreur d'interpolation ; ce qui peut être fait en manipulant l'espace des solutions approchées V_h. En effet, h représente la taille des éléments sur laquelle la solution est approximée. On remarque que V_h tend vers V, l'espace de solution de u, lorsque h tend vers zéro. Donc, si notre problème approché est défini sur un maillage constitué d'éléments infiniment petits, la solution approchée tend vers u. Malheureusement, ce problème demanderait une quantité infinie de mémoire et un laps de temps infini pour être résolu. L'adaptation de maillage permet d'optimiser l'espace des solutions approchées V_h, de taille raisonnable pour les ordinateurs actuels, et ainsi réduire l'erreur d'interpolation et tous les types d'erreurs bornés par celle-ci.

2.2 Adaptation par métrique

La distance entre deux points $\mathbf{a}$ et $\mathbf{b}$ dans l'espace euclidien $\mathbb{R}^n$ de dimension n est donnée par l'équation :

$$d(\mathbf{a},\mathbf{b}) = \sqrt{(\mathbf{b}-\mathbf{a})^T(\mathbf{b}-\mathbf{a})} \tag{2.5}$$

L'objectif de notre méthode est d'adapter le maillage en fonction de l'erreur d'interpolation. Pour cela, on va considérer une métrique anisotrope $d_{ani}(\cdot,\cdot)$ qui donne la distance euclidienne d'une transformation inversible F sur l'espace $\mathbb{R}^n$:

$$\begin{aligned} d_{ani}(\mathbf{a},\mathbf{b}) &= \sqrt{(F(\mathbf{b}-\mathbf{a}))^T F(\mathbf{b}-\mathbf{a})} \\ &= \sqrt{(\mathbf{b}-\mathbf{a})^T F^T F(\mathbf{b}-\mathbf{a})} \\ &= \sqrt{(\mathbf{b}-\mathbf{a})^T M(\mathbf{b}-\mathbf{a})} \end{aligned} \tag{2.6}$$

avec $M = F^T F$, une matrice symétrique définie positive. Mais, on voudra également étendre

notre métrique aux espaces riemanniens, dans lesquels notre transformation F est une fonction continue et lisse. Dans ce cas, pour calculer la longueur $l_M(\gamma)$ d'un chemin $\gamma(t)$ paramétré sur l'intervalle $t = [0, 1]$, on devra intégrer le long du chemin :

$$l_M(\gamma) = \int_0^1 \sqrt{\gamma\prime(t)^T M(\gamma(t))\gamma\prime(t)}dt \tag{2.7}$$

De la même manière, on peut calculer des airs et des volumes à l'aide des intégrales suivantes :

$$a_M(A) = \int_A \|(F \cdot d\mathbf{x}) \times (F \cdot d\mathbf{y})\| \tag{2.8}$$

$$v_M(V) = \int_V (F \cdot d\mathbf{x}) \times (F \cdot d\mathbf{y}) \cdot (F \cdot d\mathbf{z}) = \int_V \det(F)dv \tag{2.9}$$

On a vu à la section précédente que l'on peut calculer l'erreur d'approximation d'une solution approchée à l'aide de l'équation 2.2, où u est la solution réelle que l'on cherche à obtenir et u_h la solution discrète interpolée sur les éléments du maillage Ω_h. Le choix d'un espace fonctionnel permet de définir une cible spécifique pour la qualité du maillage. Une norme pertinente est celle de l'espace fonctionnel L^∞ tel que présenté par Labbé et al. (2011) :

$$\|u - u_h\|_{\Omega_h,\infty} = \max_{K \in \Omega_h}(\|u - u_h\|_{K,\infty}) \tag{2.10}$$

À l'aide de cette norme, les auteurs démontrent que le maillage optimal Ω_{opt} est composé des éléments K_{opt} de normes égales dans l'espace L^∞, c'est-à-dire où l'erreur d'interpolation a été équidistribuée :

$$\|u - u_h\|_{K_{opt},\infty} = \|u - u_h\|_{\Omega_{opt},\infty} \ ,\forall K_{opt} \in \Omega_{opt} \tag{2.11}$$

Dans le cas où la solution serait interpolée linéairement sur les éléments du maillage, on sait par le développement de Taylor que l'erreur d'interpolation est dominée par la dérivée seconde, soit la courbure de la solution. On utilisera la valeur absolue de la matrice hessienne de u comme mesure de la courbure. La valeur absolue M_H est obtenue en diagonalisant la matrice hessienne H, puis en prenant la valeur absolue de ses valeurs propres :

$$H = Q\Lambda Q^T \tag{2.12}$$

$$M_H = Q \mid \Lambda \mid Q^T \tag{2.13}$$

La valeur absolue de la matrice hessienne a l'avantage de pouvoir être utilisée comme tenseur métrique, puisqu'il s'agit d'une matrice symétrique définie positive. De plus, si l'on utilise des fonctions d'interpolation linéaires, les arêtes de même taille dans l'espace riemannien défini par le tenseur métrique M_H induisent des bornes similaires sur l'erreur d'interpolation.

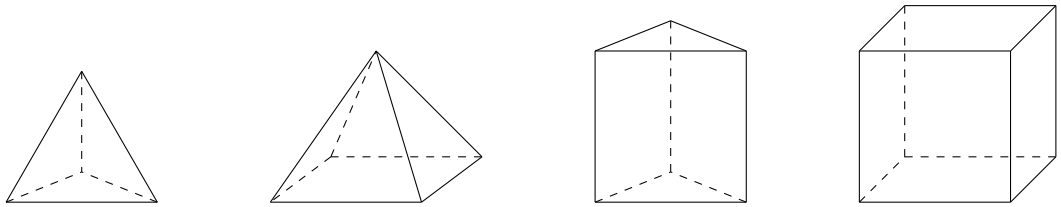
Finalement, pour adapter le maillage à notre problème, on veut équidistribuer l'erreur d'interpolation sur l'ensemble des éléments du maillage. Cela revient à trouver un maillage dont les arêtes sont de longueurs égales dans l'espace riemannien défini par la métrique M_H. De cette manière, pour un nombre d'éléments donné, on minimise l'erreur commise par notre solution approchée u_h calculée par la norme L^∞ et l'on satisfait notre critère d'optimalité du maillage.

2.3 Mesures de la qualité

Le processus d'adaptation correspond à un problème d'optimisation. On cherche à uniformiser l'erreur d'interpolation sur l'ensemble du maillage tout en imposant les contraintes suivantes : minimiser le nombre d'éléments et minimiser l'erreur maximale d'interpolation. Pour pouvoir développer des techniques de résolution adaptées à notre problème, il faut définir une mesure qui nous permettra d'évaluer l'atteinte de nos objectifs. On mesurera la qualité des éléments en se basant sur leur taille et leur forme. Dans notre contexte, un élément de bonne qualité est un élément qui répond adéquatement au niveau de précision visé par le processus d'adaptation de maillage.

Avant de présenter les mesures de qualité, il est nécessaire de définir les entités géométriques que l'on veut mesurer. D'abord, un simplexe est l'enveloppe convexe d'un ensemble de $n+1$ points dans l'espace $\mathbb{R}^n$. Cela correspond au triangle en 2D et au tétraèdre en 3D. Le simplexe est l'élément le plus simple nous permettant de remplir un espace donné. Il nous servira de base pour définir les mesures de qualité. Les éléments non simpliciaux sont tous les éléments qui possèdent plus de $n+1$ sommets. Nous nous limiterons aux pyramides (5 sommets), aux prismes (6 sommets) et aux hexaèdres (8 sommets) pour construire nos maillages en trois dimensions.

Un élément dit régulier possède des arêtes de longueurs égales. Les quatre éléments dans leur forme régulière sont présentés à la figure 2.2. Les éléments dégénérés contiennent quant à eux au moins un point où le Jacobien de leur transformation vers l'élément régulier est nul. Cependant, il est plus facile de répertorier les cas dégénérés pour les simplexes ; on abordera



(a) Tétraèdre (b) Pyramide (c) Prisme (d) Hexaèdre

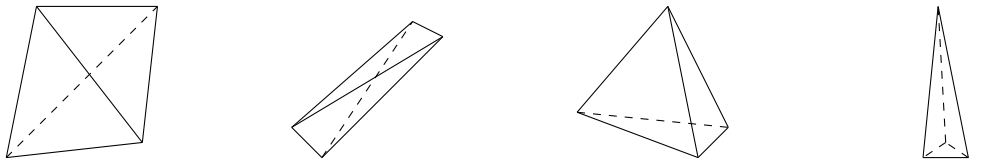
Figure 2.2 Les quatre types d'éléments tridimensionnels réguliers

ensuite le cas des éléments non simpliciaux. Un simplexe dégénéré en 3D voit son volume disparaitre, soit parce que l'ensemble de ses sommets est coplanaire, colinéaire ou fusionné en un unique point. La figure 2.3 présente quelques-uns des simplexes dégénérés les plus communs. Dompierre et al. (2005) donnent une liste exhaustive de tous les cas possibles en 2D et 3D.

Plusieurs mesures de la qualité ont été définies au fil du temps. Pour une question d'interchangeabilité, on s'assurera que toutes ces mesures aient un comportement uniforme. Entre autres, on veut que ces mesures soient maximales pour les éléments réguliers et nulles pour les éléments dégénérés. On se basera sur la définition de Dompierre et al. (2005), dérivée de Liu and Joe (1994a), pour déterminer la validité d'une mesure :

Definition 2.1. *A simplex shape measure is a continuous function that evaluates the shape of a simplex. It must be invariant under translation, rotation, reflection and valid uniform scaling of the simplex. It must be maximum for the regular simplex and it must be minimum for all degenerate simplices. For ease of comparison, it should be scaled to the interval [0, 1], and be 1 for the regular simplex and 0 for all degenerate simplices.*

Cependant, pour rendre nos algorithmes d'adaptation plus robustes, on aimerait pouvoir traiter les éléments inversés. Bien que ces éléments ne puissent faire partie d'un maillage conforme, on les rencontre souvent dans les étapes intermédiaires de nos algorithmes. Dans ce cas, il est pratique de savoir si l'on a besoin de déplacer les sommets d'un élément peu ou



(a) Sliver (b) Lamelle (c) Coin (d) Aiguille

Figure 2.3 Quelques exemples de simplexes dégénérés en 3D

beaucoup pour que celui-ci retrouve la bonne orientation. Pour y arriver, il suffit de mesurer la qualité des éléments inversés comme s'ils étaient bien orientés puis de prendre le négatif de cette mesure. Notre nouvelle échelle de qualité sera alors définie sur l'intervalle [-1, 1], où -1 désigne les éléments de forme régulière, mais inversée tandis que 0 désigne toujours les éléments dégénérés. En effet, on remarque qu'en déplaçant les sommets d'un simplexe bien orienté, celui-ci doit nécessairement passer par l'une de ses formes dégénérées avant de pouvoir être inversé et vice-versa.

Les sous-sections qui suivent présentent les mesures de qualité les plus populaires en mécanique des fluides numérique. Celles-ci sont définies pour le tétraèdre, mais peuvent éventuellement être étendues aux éléments non simpliciaux. Dompierre et al. (2005) décrivent également une procédure pour produire de nouvelles mesures. Mais, bien qu'elle permette de produire des mesures qui satisfont la définition, son utilité est limitée. Selon les auteurs, une meilleure mesure n'améliore pas un algorithme d'adaptation en soi. De plus, plus l'algorithme d'adaptation est efficace, moins la mesure choisie aurait de l'importance.

2.3.1 Angle solide minimal σ

Pour un tétraèdre $T(x_1, x_2, x_3, x_4)$, l'angle solide au sommet x_i est l'aire formée par la projection des trois autres sommets du tétraèdre sur la sphère unitaire centrée en x_i. L'angle solide minimal θ est le minimum des angles solides calculés aux quatre coins du tétraèdre :

$$\theta = \min_{1 \leq x_i \leq 4} \theta_i \tag{2.14}$$

Plutôt que de calculer θ_i à l'aide d'$arcsin(\cdot)$, on préfèrera utiliser directement $sin(\theta_i/2)$ pour diminuer les coûts de calcul. Lo (2015) propose cette formule, adaptée de Liu and Joe (1994a), pour calculer les sinus des angles solides :

$$sin(\frac{\theta_i}{2}) = \frac{\vec{a} \times \vec{b} \cdot \vec{c}}{\sqrt{2(ab + \vec{a} \cdot \vec{b})(bc + \vec{b} \cdot \vec{c})(ca + \vec{c} \cdot \vec{a})}} \tag{2.15}$$

où $a = \|\vec{a}\|$, $b = \|\vec{b}\|$, $c = \|\vec{c}\|$ et $\vec{a}$, $\vec{b}$ et $\vec{c}$ sont les vecteurs reliant le sommet x_i aux trois autres sommets du tétraèdre. On obtient finalement σ en normalisant la fonction sur l'intervalle $[0, 1]$:

$$\sigma = \frac{9}{\sqrt{6}} sin(\frac{\theta}{2}) \leq 1 \tag{2.16}$$

2.3.2 Rapport des rayons ρ

Le rapport entre les rayons des sphères inscrite r et circonscrite R du tétraèdre T est calculé à partir de son volume V et des airs A_j de ses faces :

$$r = \frac{3V}{A_1 + A_2 + A_3 + A_4} \tag{2.17}$$

$$R = \frac{\sqrt{(a+b+c)(a+b-c)(a+c-b)(b+c-a)}}{12V} \tag{2.18}$$

où a, b et c sont les produits des longueurs des arêtes opposées du tétraèdre T (un tétraèdre possède trois couples distincts d'arêtes opposées). Il ne reste plus qu'à normaliser le rapport pour obtenir une valeur dans l'intervalle $[0, 1]$:

$$\rho = \frac{3r}{R} \leq 1 \tag{2.19}$$

2.3.3 Rapport des moyennes (algébrique) η

Les mesures de qualité basées sur le rapport des moyennes font référence à la matrice de transformation F qui permet de passer du tétraèdre régulier et unitaire au tétraèdre T. Toutes les matrices assemblées lors de cette procédure sont tirées des vecteurs émanant d'un coin du tétraèdre et se terminant aux sommets adjacents. Le choix du coin est arbitraire, mais Knupp (2001) spécifie que les vecteurs doivent être ordonnés de manière à produire un volume positif suivant le système de la main droite. La transformation F est construite en rapportant d'abord un coin du cube unitaire au tétraèdre T par la transformation F_a :

$$F_a = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \mathbf{a}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_2 - \mathbf{x}_1 & \mathbf{x}_3 - \mathbf{x}_1 & \mathbf{x}_4 - \mathbf{x}_1 \end{bmatrix} \tag{2.20}$$

D'une manière analogue, on assemblera la matrice F_r permettant de transformer le même coin du cube unitaire vers le tétraèdre régulier unitaire :

$$F_r = \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 \end{bmatrix} = \begin{bmatrix} 1 & 1/2 & 1/2 \\ 0 & \sqrt{3}/2 & \sqrt{3}/6 \\ 0 & 0 & \sqrt{2/3} \end{bmatrix} \tag{2.21}$$

Ces deux matrices permettent finalement de construire la matrice de transformation F qui permet de passer du tétraèdre régulier et unitaire au tétraèdre T :

$$F = F_a \cdot F_r^{-1} \tag{2.22}$$

En se référant à la « large deformation theory », il est possible de décomposer de manière unique cette transformation en une partie de rotation pure et une autre de déformation pure :

$$\begin{cases} F = R \cdot U \\ R^T R = R R^T = I \\ U \text{ est symétrique} \end{cases} \tag{2.23}$$

En découle le tenseur de déformation Green-Cauchy C suivant :

$$C = F^T F = (RU^T) \cdot (RU) = U^T (R^T R) U = U^2 \tag{2.24}$$

Le tenseur C étant défini positif et symétrique pour les tétraèdres non dégénérés. Le rapport des moyennes est défini de manière naturelle comme étant le rapport entre la moyenne géométrique et la moyenne arithmétique des valeurs propres de la matrice C :

$$\eta = \frac{\sqrt[3]{\lambda_1 \lambda_2 \lambda_3}}{\frac{1}{3}(\lambda_1 + \lambda_2 + \lambda_3)} \leq 1 \tag{2.25}$$

On remarque qu'η atteint 0 dans le cas où au moins l'une des valeurs propres de C est nulle. Inversement, η est égale à 1 ssi $\lambda_1 = \lambda_2 = \lambda_3 = \lambda$, ce qui indique que la déformation est purement une mise à l'échelle de coefficient $\sqrt{\lambda}$:

$$C = \begin{bmatrix} \lambda_1 & & \\ & \lambda_2 & \\ & & \lambda_3 \end{bmatrix} = \begin{bmatrix} \lambda & & \\ & \lambda & \\ & & \lambda \end{bmatrix} = \lambda I = U^2 \tag{2.26}$$

On ne calculera pas les valeurs propres de la matrice C explicitement. On préfèrera passer par le calcul du déterminant et de la norme de Frobenius de la matrice de transformation F :

$$\eta = \frac{\sqrt[3]{\lambda_1 \lambda_2 \lambda_3}}{\frac{1}{3}(\lambda_1 + \lambda_2 + \lambda_3)} = \frac{3\sqrt[3]{det(C)}}{tr(C)} = \frac{3(det(F))^{2/3}}{tr(F^T F)} = \frac{3(det(F))^{2/3}}{F : F} = \frac{3(det(F))^{2/3}}{\|F\|^2} \tag{2.27}$$

2.3.4 Rapport des moyennes (géométrique) γ

Lo (1991) propose une mesure équivalente au rapport des moyennes qui peut être calculée d'une manière géométrique, c'est-à-dire qu'elle est tirée de mesures directes sur l'élément comme son volume V et la longueur de ses arêtes L_i :

$$\gamma = \frac{72\sqrt{3}V}{(\Sigma_{i=1}^{6} L_i)^{3/2}} \tag{2.28}$$

Il n'y a donc aucune référence à la matrice de transformation de l'élément. Liu and Joe (1994b) ont prouvé que $\gamma = \eta^{3/2}$.

Puisqu'on utilise dans cette formule le volume signé de l'élément, Lo (2015) affirme que cette mesure à l'avantage, par rapport aux autres mesures, de pouvoir traiter les éléments inversés. Mais il ne s'agit pas d'un avantage réel. Toutes les mesures de qualité sont capables de traiter les éléments inversés si elles disposent d'un mécanisme *a priori* pour les détecter. Par exemple, dans la version arithmétique du rapport des moyennes, on calcule déjà le déterminant de la transformation F. Ce déterminant sera négatif advenant le fait que le tétraèdre est inversé. Donc, pour rendre cette mesure apte à traiter des éléments inversés, il suffit de prendre la valeur absolue du déterminant avant de l'élever à la puissance $det(F)^{2/3}$, et de moduler la valeur finale par le signe du déterminant :

$$\eta_s = sign(det(F)) \frac{3(|det(F)|)^{2/3}}{\|F\|^2} \tag{2.29}$$

2.3.5 Conditionnement de la transformation τ

Le conditionnement de la matrice F, qui permet de passer du tétraèdre régulier au tétraèdre T, est une mesure algébrique de la qualité qui évalue l'étirement de l'élément traité. On peut calculer le conditionnement κ de la transformation à l'aide de la norme de Frobenius de la matrice de transformation F et de son inverse F^{-1} :

$$\kappa(F) = \|F\|\|F^{-1}\| \tag{2.30}$$

Une mesure basée sur ce nombre a été proposée par Knupp (1999) et il a été démontré dans Freitag and Knupp (2002) qu'elle est conforme à la définition 2.1 d'une mesure de qualité. Cette mesure est définie comme étant l'inverse du conditionnement normalisé en fonction du rang de la matrice de transformation F :

$$\tau = \frac{d}{\kappa} = \frac{3}{\|F\|\|F^{-1}\|} \tag{2.31}$$

Cette mesure a l'avantage d'avoir une interprétation bien connue en mathématique. La valeur $1/\kappa$ représente la distance entre la matrice F et l'ensemble des matrices singulières, ce qui dans notre contexte correspond aux tétraèdres dégénérés.

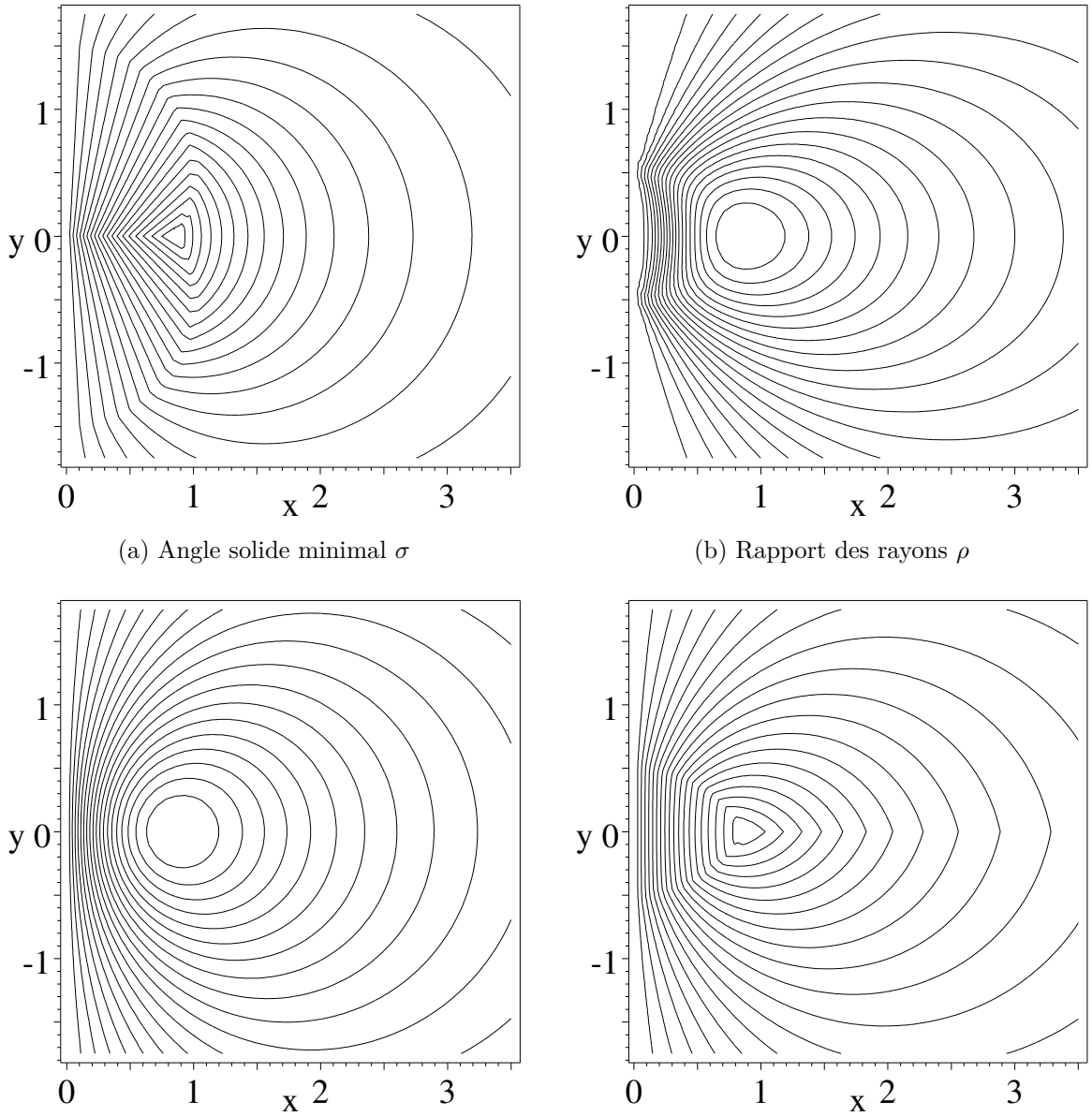
2.3.6 Équivalence des mesures

La définition 2.1 offre un cadre de comparaison pour les mesures de qualité décrites ci-dessus. Puisqu'elles démontrent le même comportement face aux mêmes types d'éléments, soit d'être maximales pour les éléments réguliers et nulles pour tous les éléments dégénérés, il est possible de démontrer que ces mesures sont toutes équivalentes. Liu and Joe (1994a) propose la relation d'équivalence suivante entre deux mesures μ et ν, où les constantes c_0, e_0, c_1 et e_1 sont strictement positives :

$$c_0\nu^{e_0} \leq \mu \leq c_1\nu^{e_1} \tag{2.32}$$

Les auteurs démontrent également que cette relation en est bien une d'équivalence puisqu'elle est réflexive, symétrique et transitive.

Bien que toutes les mesures de qualité présentées dans cette section soient équivalentes, il est intéressant de se questionner sur leur aptitude à faciliter le processus d'adaptation de maillage. En optimisation, il est bien connu que plus une fonction de coût est lisse plus la convergence sera facile et rapide, d'autant plus si elle est circulaire autour de l'optimum visé. Dompierre et al. (2005) on produit des courbes de niveau, à l'aide de la méthode introduite par Vallet (1992), pour visualiser le comportement et la régularité des différentes mesures. Deux sommets d'un triangle sont fixés aux points $(0, 1/2)$ et $(0, -1/2)$ et les courbes de niveau des différentes mesures de qualité sont tracées en fonction de la position du troisième sommet dans le plan cartésien. De la figure 2.4, on peut remarquer que toutes les mesures sont continues et qu'elles présentent des courbes de niveaux quasi circulaires au voisinage du triangle équilatéral. Cependant, seul les mesure ρ, η et τ sont différentiables partout. De plus, on observe que la mesure η affiche des courbes de niveau circulaires sur l'ensemble du domaine. Jumelé au fait qu'elle est la mesure la moins coûteuse à calculer, η semble être le choix le plus judicieux entre toutes les mesures présentées jusqu'à présent.



(a) Angle solide minimal σ

(b) Rapport des rayons ρ

(c) Rapport des moyennes η

(d) Rapport des moyennes γ

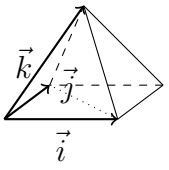
Figure 2.4 Courbes de niveau de quelques mesures de qualité, tirées de Dompierre et al. (2005) avec la permission écrite des auteurs

2.3.7 Extension aux éléments non simpliciaux

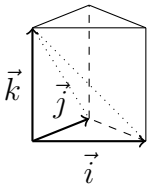
Les mesures ont été définies pour les éléments simpliciaux, mais les maillages rencontrés en mécanique des fluides numérique comportent souvent d'autres types d'éléments tels que des pyramides, des prismes et des hexaèdres. Les simplexes ont l'avantage de posséder un Jacobien (déterminant de la matrice de transformation) constant, contrairement aux éléments non simpliciaux dont le Jacobien peut varier selon le point où il est évalué ; selon Lo (2015), c'est ce qui a retardé le développement de mesures cohérentes et logiques pour les éléments non simpliciaux. Cependant, il n'est pas nécessaire de développer de nouvelles mesures pour pouvoir traiter les éléments non simpliciaux. On utilisera la même approche algébrique que celle utilisée pour η. Il suffit de remplacer les matrices de transformation F_r par les matrices de transformation des tétraèdres de référence présentées à la figure 2.5. La matrice F_a sera remplacée par la représentation matricielle des tétraèdres de références T_k, où k désigne le sommet supportant le tétraèdre de référence. La matrice de transformation finale des tétraèdres de référence est donnée par l'équation :

$$F_k = T_k \cdot F_r^{-1} \tag{2.36}$$

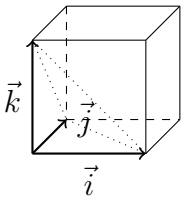
Ces tétraèdres de référence sont supportés par les sommets de valence 3 des éléments non simpliciaux. Chaque sommet sera mesuré indépendamment, puis la qualité de l'élément sera donnée par la moyenne harmonique des qualités calculées aux sommets. Notez que les py-



$$F_r = \begin{bmatrix} 1 & 0 & 1/2 \\ 0 & 1 & 1/2 \\ 0 & 0 & 1/\sqrt{2} \end{bmatrix} \tag{2.33}$$



$$F_r = \begin{bmatrix} 1 & 1/2 & 0 \\ 0 & \sqrt{3}/2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.34}$$



$$F_r = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.35}$$

Figure 2.5 Tétraèdres de référence pour les éléments non simpliciaux et les matrices F_r associées

ramides possèdent un sommet de valence 4. Celui-ci n'a pas besoin d'être évalué puisque l'évaluation des quatre autres sommets de l'élément suffit pour donner une appréciation de la qualité globale de la pyramide.

Si l'on désire obtenir une mesure étendue qui est compatible avec le rapport des moyennes dans sa version géométrique γ, il est encore possible de le faire. On a vu qu'en réalité $\gamma = \eta^{3/2}$. Pour obtenir γ_k étendue aux éléments non simpliciaux, il faut transformer le calcul d'η_k de la manière suivante :

$$\eta_k = \frac{3det(F_k)^{2/3}}{\|F_k\|^2} \tag{2.37}$$

$$\gamma_k = \frac{3\sqrt{3}det(F_k)}{\|F_k\|^3} \tag{2.38}$$

Une fois de plus, on souhaitera ajouter de la robustesse à notre méthode dans le cas où l'un des sommets produirait un tétraèdre de référence inversé. Dans ce cas, on obtiendra une qualité négative à l'endroit de ce sommet, ce qui compromettrait le calcul de la moyenne harmonique. C'est pourquoi, si au moins un des sommets est renversé, la qualité de l'élément dans sa globalité sera égale à celle de son pire sommet, ce qui donnera nécessairement une qualité négative pour l'élément. Le calcul final pour chacun des types d'éléments est donné par les équations 2.39, 2.40, 2.41 et 2.42, où $\alpha(\cdot)$ est l'une des mesures de qualité.

$$\alpha(Tetraèdre) = \alpha_i, \text{ pour i} \in 1..4 \text{ quelconque} \tag{2.39}$$

$$\alpha(Pyramide) = \begin{cases} \dfrac{4}{\sum_{i=1..4} \frac{1}{\alpha_i}} & \text{si } \alpha_i \geq 0, \forall i = 1..4 \\ \min\limits_{i=1..4} \alpha_i & \text{sinon} \end{cases} \tag{2.40}$$

$$\alpha(Prisme) = \begin{cases} \dfrac{6}{\sum_{i=1..6} \frac{1}{\alpha_i}} & \text{si } \alpha_i \geq 0, \forall i = 1..6 \\ \min\limits_{i=1..6} \alpha_i & \text{sinon} \end{cases} \tag{2.41}$$

$$\alpha(Hexaèdre) = \begin{cases} \dfrac{8}{\sum_{i=1..8} \frac{1}{\alpha_i}} & \text{si } \alpha_i \geq 0, \forall i = 1..8 \\ \min\limits_{i=1..8} \alpha_i & \text{sinon} \end{cases} \tag{2.42}$$

2.3.8 Généralisation des mesures aux espaces riemanniens

Telles quelles, les mesures de qualité σ, ρ, η, γ et κ ne présentent pas une grande utilité en mécanique des fluides numérique. Elles permettent de produire des maillages « réguliers » formés d'éléments peu étirés certes, mais ne tiennent aucunement compte de l'erreur d'approximation que l'on tente de réduire par l'adaptation de maillage. Tout particulièrement, aucune mention n'a encore été faite de la fonction métrique qui devrait normalement prescrire la taille et la forme des éléments.

Pour généraliser les mesures de qualité aux espaces riemanniens, on introduira la notion de métrique dans la définition des mesures. Une métrique permet naturellement de mesurer des grandeurs géométriques, telles que la longueur, l'aire et le volume, dans un espace riemannien. Heureusement, les mesures d'angle solide minimal σ, de rapport des rayons ρ et la qualité γ sont déjà exprimées en fonction de ces grandeurs géométriques. Ces mesures permettront de produire des éléments réguliers dans l'espace riemannien, bien que, vu de l'espace euclidien, ils paraissent étirés/écrasés. Cette modification ajoute du contrôle sur la forme des éléments, mais ne permet toujours pas de contrôler leur taille.

Lo (2015) mentionne qu'il n'est pas nécessairement possible de construire un élément dit régulier pour un espace riemannien donné. Un élément qui possède des arêtes de longueurs unitaires peut ne pas atteindre la qualité maximale de 1, suivant le comportement de la fonction métrique à l'intérieur de l'élément. De plus, il serait impossible de prouver que les mesures de qualité possèdent un seul maximum lorsque l'on introduit une métrique riemannienne dans leur définition. L'auteur ajoute que dans le cas des maillages anisotropes, puisque la définition d'un élément régulier n'a plus de sens, une plus grande importance devrait être accordée aux distances qu'à la forme. C'est ce qui le motive à proposer la mesure de qualité combinée λ, bien moins coûteuse en temps de calcul que les mesures classiques généralisées aux espaces riemanniens. Celle-ci est formée d'une mesure de qualité classique calculée dans l'espace euclidien, modulée par la conformité des longueurs des arêtes à la métrique δ_i :

$$\delta_i = \min(\frac{r_i}{\rho_i}, \frac{\rho_i}{r_i}) \tag{2.43}$$

où r_i et ρ_i sont respectivement la longueur actuelle et la longueur cible de l'arête i dans la métrique. La mesure combinée, dans le cas du tétraèdre, est alors :

$$\lambda = \alpha\delta_1\delta_2\delta_3\delta_4\delta_5\delta_6 \tag{2.44}$$

α étant l'une des mesures calculée dans l'espace euclidien. Il est également affirmé par Lo

(2015) qu'il serait sensé d'évaluer la forme α dans l'espace euclidien puisqu'un élément de bonne forme ne peut être que légèrement distortionné sous l'influence d'une métrique lisse. Toutefois, je ne sais pas si cette affirmation tient compte des éléments des couches limites fréquemment rencontrées en mécanique des fluides numérique. Ceux-ci sont généralement écrasés le long des frontières, leur donnant de très grands rapports d'aspect. Bien qu'ils soient bien proportionnés du point de vue de la métrique riemannienne, aucune mesure classique ne leur attribuerait une qualité élevée dans l'espace euclidien.

Pour remédier aux problèmes du manque de contrôle sur la taille des éléments des mesures classiques et inversement du manque de contrôle sur la forme des éléments de la mesure combinée λ, Labbé et al. (2004) propose une ultime mesure de la qualité : la non-conformité à la métrique. Celle-ci est entièrement basée sur la notion de métrique et s'étend facilement, à la manière d'η, aux éléments non simpliciaux. Elle constitue de ce fait la mesure de qualité de choix dans le cadre de la présente recherche.

2.3.9 Conformité et non-conformité à la métrique

Labbé et al. (2004) décrivent la non-conformité à la métrique de la manière suivante :

> *This measure is a unique, dimensionless number which not only completely characterizes, both in size and in shape, a single simplex, but can also be used for a whole mesh in two or three dimensions, coarse or fine, independently of the size of the domain, generated for an isotropic or an anisotropic size specification map.*

Cette mesure s'étend particulièrement bien aux éléments non simpliciaux. Mais avant de passer à sa définition, il est souhaitable de faire la distinction entre deux types de métriques.

D'une part, il y a la métrique de contrôle M_S telle que présentée à la section 2.2. Elle est fournie en entrée du processus d'adaptation de maillage sous la forme d'une carte de taille. Elle est générée par l'estimateur d'erreur *a posteriori* d'après la courbure de la solution approchée. C'est cette métrique qui permet de mesurer les dimensions des éléments dans l'espace de contrôle.

D'autre part, la métrique d'un élément M_K est le tenseur métrique de la transformation permettant de ramener l'élément à sa forme régulière équilatérale. C'est effectivement cette métrique qui est utilisée dans la définition d'η. On se rappellera que cette métrique est constante sur tout l'élément dans le cas des simplexes, mais qu'elle varie pour les éléments non simpliciaux en fonction du point où elle est évaluée. La solution la plus simple est d'assigner une métrique à chaque sommet de l'élément non simplicial qui peut soutenir un simplexe de référence.

L'ensemble des métriques d'élément forme une fonction constante par morceaux. Pour pouvoir comparer ces métriques à la carte de taille, qui est continue et lisse, on devra avoir recours à une forme de moyennage. Les éléments du maillage couvrent chacun une partie de l'espace. On calculera la moyenne de la métrique de contrôle $\overline{M_S}(K)$, pour un élément K, par l'intégrale suivante :

$$\overline{M_S}(K) = \int_K M_S(\mathbf{x})dk \Big/ \int_K dk \tag{2.45}$$

que l'on calculera en pratique à l'aide d'une quadrature de Gauss. Il devient maintenant possible de faire correspondre une métrique de contrôle $\overline{M_S}(K)$ avec une métrique d'élément M_K pour chaque élément K. Si l'élément était régulier et équilatéral dans l'espace de contrôle, l'égalité suivante serait observée :

$$\overline{M_S}(K) = M_K \tag{2.46}$$

Puisque ce n'est généralement pas le cas, on aimerait mesurer la distance entre ces deux métriques ; distance qui nous informerait sur la conformité de l'élément à la carte de taille imposée. En cas d'inégalité, on peut calculer deux formes de résidus, l'un plus sensible lorsque le volume de l'élément tend vers 0 (R_s) et l'autre plus sensible lorsque sont volume tend vers l'infini (R_b). Le résidu total R_t est la somme de ces deux résidus :

$$R_s = \overline{M}_S^{-1} M_K - I \tag{2.47}$$

$$R_b = M_K^{-1} \overline{M}_S - I \tag{2.48}$$

$$R_t = R_s + R_b = \overline{M}_S^{-1} M_K + M_K^{-1} \overline{M}_S - 2I \tag{2.49}$$

Finalement, le coefficient de non-conformité à la métrique $\mathscr{E}$ est défini comme étant la norme euclidienne du résidu total :

$$\mathscr{E}_K = \|R_t\| = \sqrt{tr(R_t^T R_t)} \tag{2.50}$$

On obtient alors un coefficient nul pour les éléments respectant parfaitement la carte de taille et un coefficient infini pour tous les éléments dégénérés. Bien que ce comportement ne

corresponde pas à la définition 2.1 d'une mesure de la qualité, il est possible de dériver une mesure à partir de ce coefficient que l'on nommera *conformité à la métrique* $\mathscr{C}$:

$$\mathscr{C}_K = 1/(1 + \mathscr{E}_K) \tag{2.51}$$

Cette mesure donne bien 1 pour les éléments réguliers dans l'espace de contrôle et 0 pour tous les éléments dégénérés. Encore une fois, il est possible de supporter les éléments inversés en modulant la mesure par le signe du volume.

2.3.10 Qualité d'un maillage

Mesurer la qualité des éléments sert ultimement à déterminer la qualité d'un maillage dans son ensemble. Il n'existe pas de manière formelle de définir la qualité d'un maillage. Les deux mesures que l'on observe le plus souvent dans la littérature sont la qualité du pire élément α_{min} et la qualité moyenne α_{mean}. Bien que la qualité moyenne semble *a priori* être un bon indicateur de la qualité globale d'un maillage, on peut aussi supposer qu'à l'instar d'une chaîne, sa qualité dépende fortement de son élément le plus faible. Par exemple, dans la MEF, le conditionnement de la matrice de rigidité est dominé par le conditionnent du pire élément (voir Shewchuk (2002a)). Quelques solveurs itératifs sont capables de contourner ce problème, mais dans la plupart des cas, un conditionnement élevé de la matrice de rigidité implique une erreur de discrétisation élevée sur tout le domaine. Dans cette recherche, pour comparer les maillages adaptés, on utilisera systématiquement le minimum α_{min} ainsi que la moyenne harmonique α_{harm} des qualités, la moyenne harmonique étant plus sensible que la moyenne arithmétique aux éléments de faible qualité.

2.4 Algorithmes d'adaptation

Maintenant que l'on sait mesurer la qualité d'un maillage et de ses éléments, on peut se pencher sur les moyens nous permettant d'augmenter cette qualité. L'adaptation de maillage n'est pas la seule façon d'y parvenir. On pourrait entièrement remailler le domaine dans l'espace riemannien. On procéderait dans ce cas par triangulation Delaunay, par avancement de front ou une combinaison des deux. Inversement, en choisissant d'adapter le maillage, on part du principe que le maillage original est une bonne approximation du maillage optimal. Ainsi, on suppose qu'en modifiant localement la position des nœuds et leurs connectivités, on devrait rapidement minimiser et uniformiser l'erreur d'interpolation.

L'adaptation de maillage est essentiellement un problème d'optimisation. Le but étant de

distribuer l'erreur d'interpolation uniformément sur l'ensemble des éléments du maillage, la mesure de qualité choisie constitue la fonction coût de notre problème. Deux grandes familles d'algorithmes permettent de distinguer les algorithmes d'adaptation.

La première famille regroupe les techniques de déplacement de nœuds. Celles-ci ne changent ni le nombre de nœuds, ni le nombre d'éléments, ni la manière dont les nœuds sont connectés entre eux. Leur seule préoccupation est de trouver la position idéale de chaque nœud pour une topologie donnée. Ce sont les algorithmes que l'on tentera de paralléliser sur GPU dans cette recherche.

La deuxième famille regroupe les techniques de modifications topologiques. Celles-ci changent le nombre de nœuds, le nombre d'éléments et la manière dont les nœuds sont connectés. Elles peuvent éventuellement modifier la position des nœuds, mais il s'agit généralement d'un effet de bord. Les modifications topologiques seront abordées succinctement dans ce mémoire. Elles ne seront pas parallélisées sur GPU et dépassent donc légèrement le cadre de cette recherche.

Comme Freitag and Ollivier-Gooch (1997) et Lo (1997) nous le font remarquer, les deux familles d'algorithmes sont complémentaires. Les algorithmes de déplacement de nœuds ont généralement une efficacité limitée puisqu'ils sont incapables de raffiner ou de déraffiner un maillage. Un maillage trop grossier n'arrivera jamais à réduire suffisamment l'erreur d'approximation même si ses nœuds sont positionnés de manière optimale. Inversement, un maillage très fin permettra d'obtenir une faible erreur d'approximation, mais il contiendra plus d'éléments que nécessaire ce qui allongera le temps de résolution inutilement. D'un autre côté, les modifications topologiques opèrent dans un espace d'optimisation discret. Pour une configuration donnée de nœuds, il existe un nombre fini de façons de les connecter. Les algorithmes de déplacement de nœuds opèrent quant à eux dans un espace continu, donc infiniment plus vaste. Une autre façon d'exprimer la complémentarité des deux familles est de dire que l'un régularise la topologie tandis que l'autre régularise la géométrie du maillage.

2.4.1 Lissage par déplacement de nœuds

Les algorithmes de déplacement de nœuds ne manipulent pas directement les éléments, mais plutôt les nœuds qui sont partagés par plusieurs éléments. Le voisinage élémentaire associé à un nœud est le polyèdre formé des éléments qui englobent ce nœud (voir figure 2.6). Tous les algorithmes de déplacement de nœuds se basent sur la qualité de ce voisinage élémentaire pour déterminer la position idéale du nœud qu'ils traitent. Nous aborderons dans cette section les algorithmes les plus utilisés en adaptation de maillage. L'historique des techniques majeures de déplacement de nœuds peut être consulté dans Lo (2015).

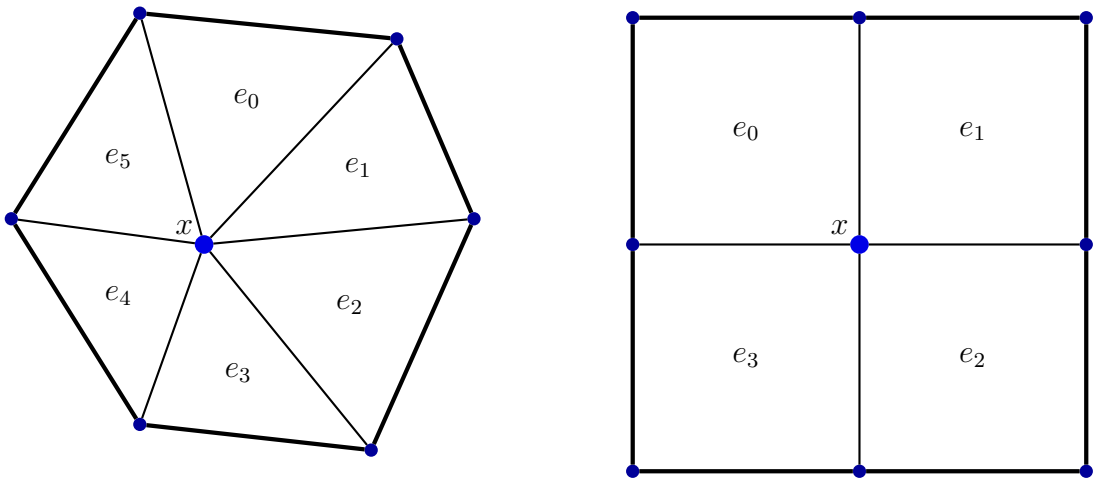


Figure 2.6 Exemples de voisinages élémentaires en deux dimensions

2.4.2 Qualité d'un voisinage élémentaire

Les algorithmes d'adaptation ont besoin d'une fonction coût pour pouvoir positionner de manière optimale les nœuds du maillage. Jusqu'ici, nous avons défini des mesures permettant de mesurer la qualité des éléments pris individuellement. Toutefois, lorsqu'on tente d'augmenter la qualité d'un élément en particulier, cela peut souvent détériorer la qualité de ses voisins. C'est pourquoi on préfère se concentrer sur les nœuds du maillage pour développer nos algorithmes de déplacement. Un nœud est généralement partagé par plusieurs éléments. Ces éléments forment un *voisinage élémentaire*. Étant donné un maillage $\mathscr{M} = \{v_i, i = 1..M, e_k, k = 1..N\}$ de N éléments, ce voisinage élémentaire est un polyèdre P ayant pour centre le nœud x :

$$P(x) = \{e_k \in \mathscr{M}; x \in e_k\} \tag{2.52}$$

où e_k sont les éléments du voisinage élémentaire et $x \in e_k$ signifie que le nœud x est l'un des sommets de l'élément e_k. La fonction coût des algorithmes sera définie à partir des qualités des éléments du voisinage élémentaire. On peut construire la fonction coût $\alpha_P(x)$ de différentes manières. Pour cette recherche, il a été jugé souhaitable que cette fonction soit capable de traiter les éléments inversés et qu'elle soit la plus lisse possible. Malheureusement, ces deux contraintes semblent difficilement compatibles. Un compromis a été trouvé par l'utilisation, dans le cas où tous les éléments du voisinage élémentaire sont bien orientés, de la moyenne harmonique des qualités et, dans le cas où au moins un élément est inversé, du minimum des qualités :

$$\alpha_P(x) = \alpha(P(x)) = \begin{cases} \dfrac{n}{\sum_{e_k \in P(x)} \alpha(e_k)} & \text{si } \alpha(e_k) \geq 0,\ \forall e_k \in P(x) \\ \min\limits_{e_k \in P(x)} \alpha(e_k) & \text{sinon} \end{cases} \tag{2.53}$$

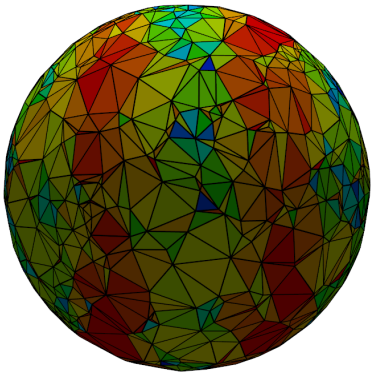
où n est le nombre d'éléments dans le voisinage élémentaire. La fonction coût ainsi définie conserve la continuité et le caractère lisse de la mesure choisie, tant qu'un élément du voisinage n'est pas inversé. On reconnaît la construction des mesures de qualité pour les éléments non simpliciaux (voir équations 2.39 à 2.42). Si la qualité d'un voisinage élémentaire était donnée par la qualité de son pire élément, on aurait obtenu des discontinuités dans les dérivées. Celles-ci n'empêcheraient pas les algorithmes d'optimisation basés sur le gradient de fonctionner, mais elles ralentiraient leur taux de convergence.

Lissage laplacien

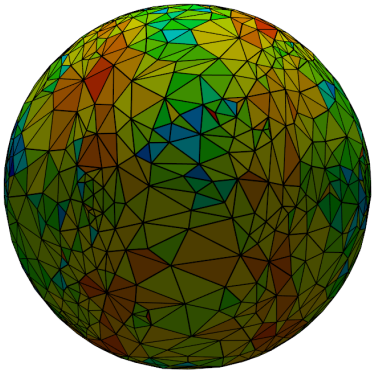
Avant l'élaboration des mesures de qualité classiques, Herrmann (1976) proposait une première technique pour régulariser la position des nœuds : le lissage laplacien (*Laplace Smoothing*). Celle-ci consiste à déplacer un nœud x au centre de son voisinage élémentaire c_P. L'équation 2.54 présente le calcul du centre comme la moyenne des positions P_i des nœuds du voisinage élémentaire. Si les arêtes qui émanent du nœud étaient des ressorts, le centre correspondrait au point d'équilibre du système. Cependant, il est généralement admis que cette heuristique donne de mauvais résultats en pratique. En trois dimensions, on se retrouve souvent avec des éléments inversés à la fin du processus, et ce même pour des maillages originalement conformes. La figure 2.7 donne une idée du nombre de tétraèdres qui peuvent être inversés suite à l'application du lissage laplacien sur une triangulation Delaunay en 3D.

$$c_P = \sum_{i=1}^{n} \frac{1}{n} P_i \tag{2.54}$$

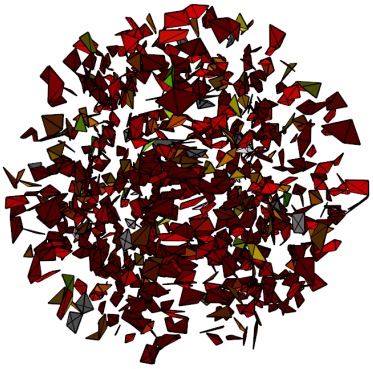
Pour éviter de dégrader le maillage davantage, Lo (1997) propose une modification à l'algorithme, ne déplaçant un nœud que si sa nouvelle position augmente la qualité de son voisinage élémentaire. Cette nouvelle version de l'algorithme est nommée lissage laplacien de qualité (*Quality Laplace)*. Formellement, l'algorithme consiste à trouver le centre c_P du voisinage élémentaire P, à tracer une droite d passant par le nœud x et le centre et finalement à maximiser la qualité du voisinage élémentaire en déplaçant le nœud x le long de la droite d. En pratique, on se limitera à tester moins d'une dizaine de positions aux alentours de x et c_P. L'algorithme est illustré à la figure 2.8 pour un maillage triangulaire.



(a) Initial



(b) Adapté (élém. valides)



(c) Adapté (élém. inversés)

Figure 2.7 Le lissage laplacien adapte le maillage, mais produit des éléments inversés

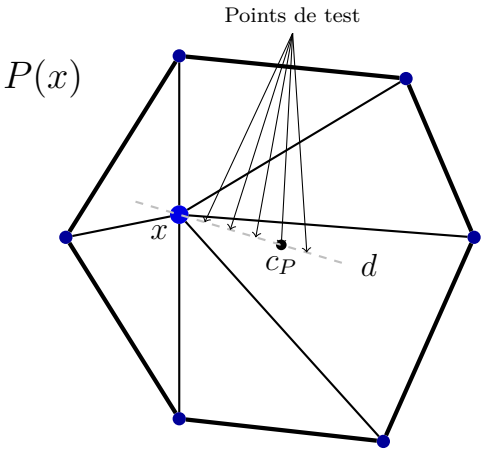


Figure 2.8 Avec le laplacien de qualité, seule la position maximisant la qualité du voisinage élémentaire est retenue.

Descente du gradient

La descente du gradient est l'une des trois méthodes dites d'« optimisation locale » qui seront présentées dans ce mémoire. Il s'agit de la méthode classique consistant d'abord à calculer le gradient de la qualité du voisinage élémentaire à la position du nœud x, puis à déplacer le nœud dans le sens du gradient afin d'augmenter la qualité du voisinage élémentaire. Notez bien que l'on monte le long du gradient puisqu'il s'agit ici d'un problème de maximisation. Le processus d'optimisation est schématisé à la figure 2.9.

À la manière du lissage laplacien, on testera plusieurs positions le long du gradient en retenant celle qui maximise la qualité du voisinage élémentaire. On peut effectuer plusieurs itérations sur le même nœud avant de passer au prochain ; chaque itération donnant une meilleure approximation de la direction à prendre pour rejoindre le maximum local.

Il est bien connu que ce type d'algorithme nécessite une fonction lisse pour donner de bons résultats. C'est précisément la raison pour laquelle on a pris soin de construire les mesures de qualité des éléments non simpliciaux α et des voisinages élémentaires α_P à l'aide de moyennes.

Nelder-Mead

Une autre manière commune de définir $\alpha_P(x)$ est de retourner la qualité du pire élément du voisinage élémentaire :

$$\alpha_m(x) = \min_{e_i \in P(x)} (\alpha(e_i)) \tag{2.55}$$

De cette manière, la fonction coût à maximiser n'est lisse que par morceaux. Si l'on avait choisi cette définition pour mesurer la qualité d'un voisinage élémentaire, il aurait été conseillé d'avoir recours à un algorithme spécialement conçu pour maximiser les fonctions non lisses. L'algorithme Nelder-Mead (Conn et al. (2009)) permet d'optimiser une fonction scalaire $f(\vec{x}) \to \mathbb{R}$ à plusieurs variables sans avoir recours à la dérivée de la fonction. Elle a déjà été utilisée dans le cadre d'optimisation de maillage par Park and Shontz (2010) et Cheng et al. (2015).

Avant d'amorcer l'algorithme, il faut construire un simplexe de recherche pour se déplacer dans l'espace des variables. Puisque nous travaillons en 3D, on choisira un tétraèdre à angle droit dont l'un des sommets est sur le nœud x et dont les arêtes qui en émanent sont parallèles au système d'axes. On s'assurera aussi de mettre ce simplexe à l'échelle en fonction du rayon du voisinage élémentaire. On assigne une valeur à chacun des sommets du simplexe de recherche, soit la qualité que le voisinage élémentaire aurait si le nœud x était positionné à

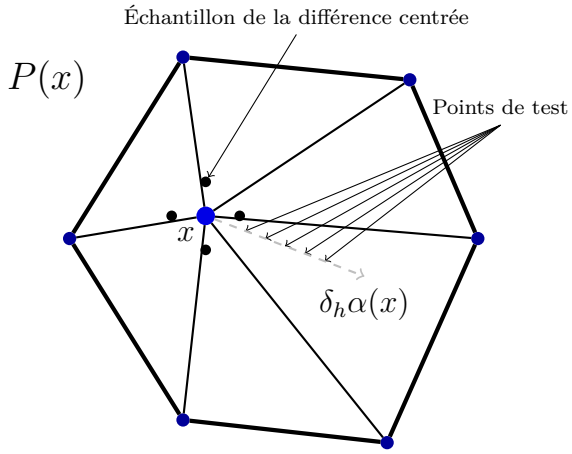


Figure 2.9 On monte le long du gradient par recherche linéaire

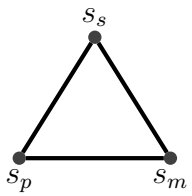
leur endroit.

L'algorithme débute en sélectionnant le sommet du simplexe qui détient la pire qualité. Par une succession de réflexions, dilatations et contractions (nommées *reflect*, *expand* et *contract* en anglais), il tente de trouver une meilleure position pour ce sommet. Une quatrième opération rapproche tous les sommets du simplexe vers le meilleur sommet par une opération de rétrécissement (*shrink*) dans le cas où les opérations de contraction échoueraient à approcher le maximum local. En répétant le processus plusieurs fois, le simplexe devrait se déplacer vers un maximum local, puis l'entourer, rétrécir et éventuellement se confondre avec lui. On termine l'algorithme lorsqu'une des trois conditions suivantes est vérifiée : le simplexe est suffisamment petit (convergence du domaine), les valeurs du pire et du meilleur sommet sont suffisamment proches (convergence des résultats) ou le nombre d'itérations dépasse un certain seuil (non-convergence de la méthode).

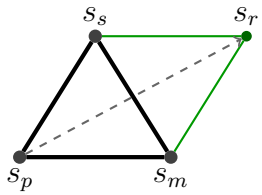
Les quatre opérations sont illustrées en 2D à la figure 2.10, où s_p est le pire sommet, s_m est le meilleur et s_s est le second meilleur. Une explication plus approfondie de la méthode est donnée par Singer and Nelder (2009). Bien sûr, la fonction $\alpha_P(x)$ telle que définie à la section 2.4.2 est suffisamment lisse pour être optimisée par l'algorithme du gradient.

Force brute

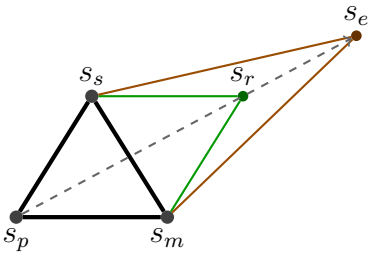
Les algorithmes précédents tentent de trouver la position optimale du nœud x en échantillonnant la fonction $\alpha_P(x)$ en un petit nombre de positions. Par exemple, l'algorithme du gradient détermine l'axe de recherche en approximant le gradient de $\alpha_P(x)$ par une différence centrée. Nelder-Mead travaille avec un simplexe de recherche pour trouver les prochaines



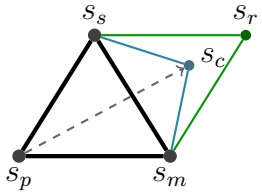
(a) *Départ*



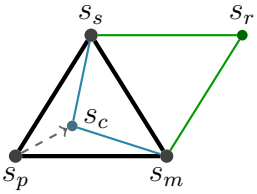
(b) *Reflect* s_r



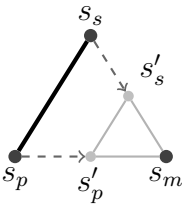
(c) *Expand* s_e



(d) *Contract outside* s_c



(e) *Contract inside* s_c



(f) *Shrink*

Figure 2.10 Les quatre déplacements possibles de l'algorithme Nelder-Mead

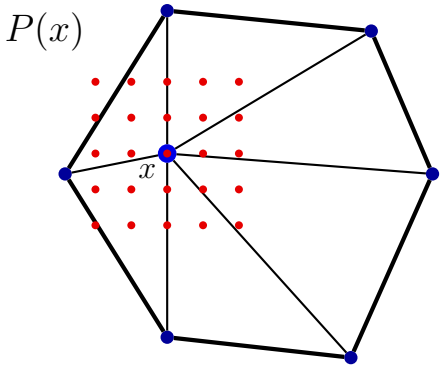


Figure 2.11 Par force brute, on échantillonnera un grand nombre de positions autour de la position actuelle de x

positions à tester et éventuellement se rapprocher d'un maximum local. Toutefois, si l'on veut paralléliser le processus d'optimisation local au maximum, il pourrait être avantageux de tester plus de positions tout en laissant tomber la recherche de direction vers l'optimum.

L'algorithme de force brute consiste simplement à tester un grand nombre de positions autour du nœud x et à déplacer le nœud là où la qualité du voisinage élémentaire s'est montrée maximale. La distribution des échantillons est arbitraire. Une distribution 2D en forme de grille est donnée à la figure 2.11. Cet algorithme est lent sur le CPU, mais, contrairement aux lissages laplaciens, converge vers le même optimum que les autres algorithmes d'optimisation locale. Il est beaucoup plus lent que les autres algorithmes, car il requiert un grand nombre d'échantillons même pour une grille de recherche modeste : 64 échantillons pour une configuration 4x4x4 en 3D. Cependant, il sera démontré au chapitre 5 qu'il possède un grand potentiel de parallélisation sur GPU.

2.4.3 Modifications topologiques

Contrairement aux algorithmes de déplacement de nœuds, les opérations topologiques permettent de raffiner et de déraffiner le maillage. De plus, elles permettent de changer la valence des nœuds. En deux dimensions, dans l'espace euclidien, un maillage divisé régulièrement à l'aide de triangles équilatéraux possède des nœuds intérieurs de valence 6 comme on peut le voir à la figure 2.12. En trois dimensions, contrairement à ce que Aristote croyait (Struik (1925)), il est impossible de diviser régulièrement l'espace en tétraèdres équilatéraux, mais l'expérience montre que les nœuds intérieurs d'un maillage tétraédrique ont en moyenne une valence de 15. La figure 2.13 montre un nœud au milieu d'un voisinage élémentaire icosaédrique, ce qui lui donne une valence de 12. Remarquez que, bien que la surface du voisinage élémentaire soit un polyèdre régulier, les tétraèdres qui le composent ne sont pas équilatéraux.

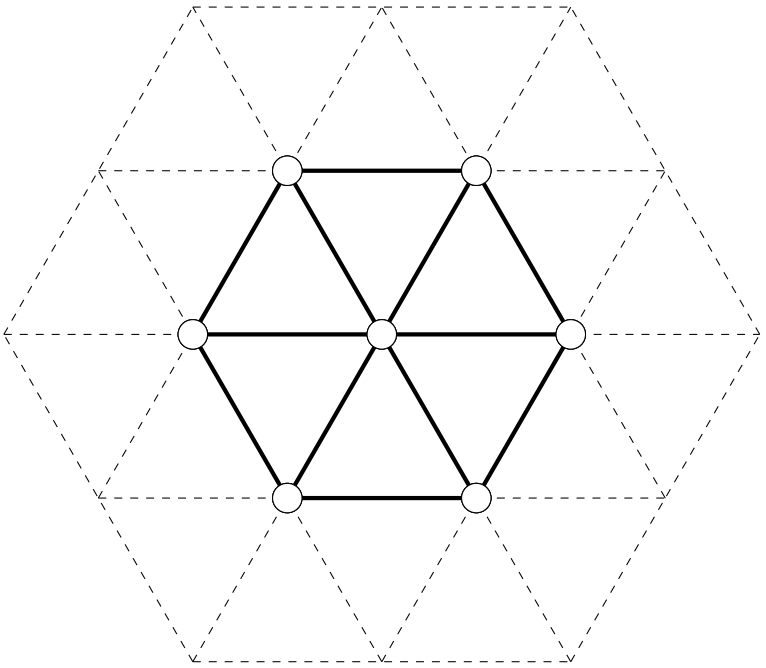


Figure 2.12 Partie intérieure d'un maillage triangulaire régulier

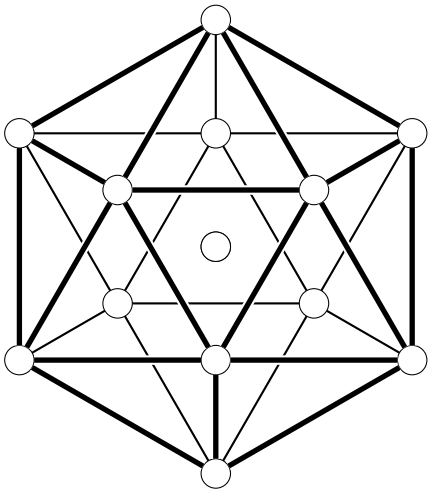


Figure 2.13 Nœud de valence 12 au sein d'un voisinage élémentaire icosaédrique

Il existe trois manières de raffiner un maillage par l'ajout de nœuds. On peut soit ajouter un nœud à l'intérieur d'un élément, soit sur une face ou soit sur une arête. Le meilleur choix dépend de la configuration des éléments en place. Ajouter un nœud à l'intérieur d'un tétraèdre est une bonne manière de réduire la valence des nœuds, puisque le nœud inséré aura nécessairement une valence de 4.

Il existe trois façons de déraffiner un maillage par fusion de nœuds. L'effondrement d'arête supprime l'arête qui relie deux nœuds voisins et par la même occasion l'anneau de tétraèdres qui entouraient cette arête. L'effondrement de face fusionne tous les nœuds appartenant à une même face. Finalement, l'effondrement de volume fusionne les nœuds d'un élément. La fusion de nœuds peut produire un nœud de très grande valence puisque le nœud résultant hérite des voisins des tous les nœuds fusionnés.

Il est aussi possible d'uniformiser la valence des nœuds sans ajouter ou supprimer de nœuds. On procédera par retournement d'arêtes et retournement de faces ; deux techniques antagonistes qui reconfigurent localement un ensemble de tétraèdres. Le retournement de faces permet de connecter par une arête deux nœuds qui étaient séparés par une mosaïque de triangles. Inversement, le retournement d'arêtes permet de séparer par une mosaïque de triangles deux nœuds qui étaient connectés par une arête. Les deux opérations sont illustrées à la figure 2.14 où deux tétraèdres séparés par la face $\triangle v_2v_3v_4$ deviennent un anneau de trois tétraèdres autour de l'arête $\overline{v_1v_5}$. Une description détaillée de ces retournements est donnée par Freitag and Ollivier-Gooch (1997), Shewchuk (2002b) et Klingner and Shewchuk (2008).

2.5 Adaptation en parallèle

Les solveurs du type MEF et MVF ont bénéficié d'avancées importantes dans les dernières années en ce qui concerne la parallélisation des méthodes sur des ordinateurs multicœurs et des systèmes distribués. Certaines recherches, telles que Corrigan and Löhner (2011), se penchent également à la réécriture de solveurs pour des systèmes distribués multi-GPU.

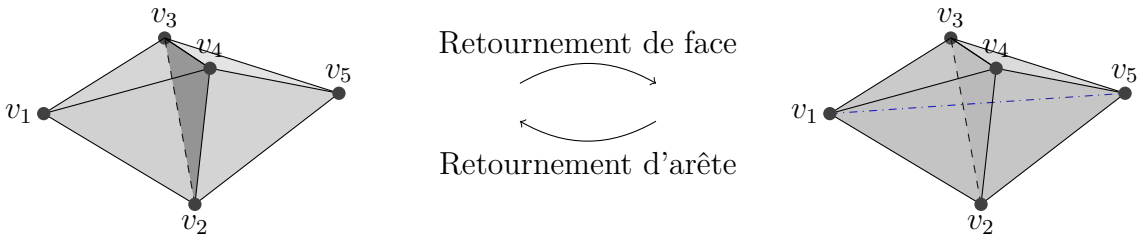


Figure 2.14 Le retournement de face est l'opération opposée au retournement d'arête

Par contre, beaucoup moins d'efforts ont été investis dans la parallélisation des méthodes d'adaptation de maillage, surtout dans le domaine du calcul généralisé sur GPU (GPGPU). Pourtant, on observe en pratique que le temps alloué à l'adaptation de maillage peut dépasser celle de la résolution. On s'attend à ce que cet écart augmente davantage si de nouvelles méthodes d'adaptation en parallèle ne sont pas proposées.

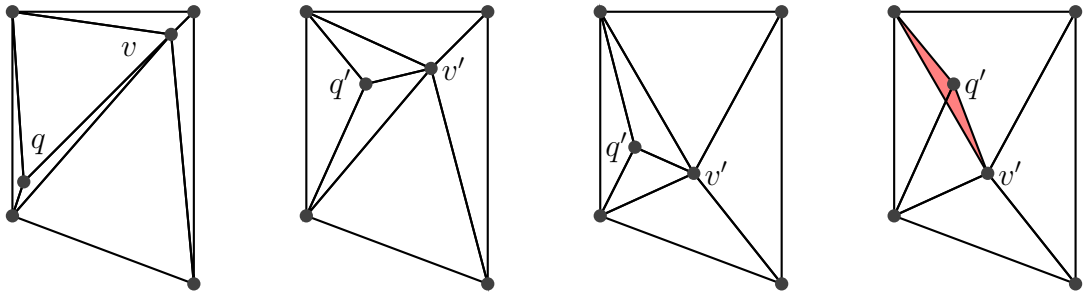
Cette recherche s'attarde principalement à la parallélisation des algorithmes locaux de déplacement de nœuds présentés à la section 2.4.1. Ces algorithmes possèdent les trois qualités nécessaires pour être parallélisables *a priori* sur le GPU : manipuler une grande quantité de données, traiter uniformément celles-ci et minimiser les dépendances entre fils d'exécution. Il existe aussi des algorithmes globaux de déplacement de nœuds, mais ceux-ci ont été mis de côté puisqu'ils bénéficient directement des avancées acquises dans la résolution parallèle des systèmes d'équations linéaires. Quant aux modifications topologiques, c'est la mise à jour des dictionnaires de connectivité qui les rend mals adaptés au GPU (voir D'Amato and Vénere (2013)).

On verra dans ce chapitre le principe fondamental à respecter pour développer des algorithmes parallèles de déplacement de nœuds valides. Ce principe a été appliqué à l'origine pour développer des versions parallèles et distribuées, mais on verra qu'il est aussi applicable dans le développement d'algorithmes sur GPU. On explorera ensuite le modèle d'exécution parallèle et l'architecture des GPU. La connaissance de leur architecture est primordiale pour comprendre les avantages et les limites de ce type de processeur. Puis, on présentera quelques implémentations d'algorithmes de déplacement de nœuds sur GPU trouvées dans la littérature qui se rapprochent des objectifs de la présente recherche.

2.5.1 Ensembles de nœuds indépendants

Les fondements du déplacement de nœuds en parallèle ont été développés par Freitag et al. (1999). En partant du constat que la position finale des nœuds lissés n'est pas unique et que celle-ci dépend de l'ordre dans lequel les nœuds sont traités, les auteurs ont défini une règle assurant l'exécution correcte de n'importe quel algorithme de déplacement de nœuds en parallèle sans l'aide de primitives de synchronisation. La non-unicité des positions des nœuds lissés est illustrée aux figures 2.15. Ce phénomène peut mener à la génération d'éléments inversés si deux sommets voisins sont déplacés simultanément, tel que montré à la figure 2.15 c). On en conclut que lors du déplacement d'un nœud, tous les nœuds faisant partie de son voisinage élémentaire doivent rester immobiles.

Une manière de déterminer les ensembles de nœuds pouvant être déplacés simultanément est de procéder par coloriage de graphe. On doit étendre la procédure donnée par Rokos



(a) Initial (b) q avant v (c) v avant q (d) Simultanément

Figure 2.15 L'ordre des nœuds à un impact sur la qualité finale et potentiellement sur la conformité du maillage si certains d'entre eux sont déplacés simultanément

et al. (2011) et Cheng et al. (2015) puisqu'elle n'est valide que pour les maillages constitués d'éléments simpliciaux. Les nœuds du graphe à colorier correspondent directement aux nœuds du maillage. On ajoute une arête entre chaque paire de nœuds qui appartiennent à un même élément. Par exemple, si notre maillage est formé d'un unique hexaèdre, on retrouvera une arête entre chacun de ses nœuds (soit $\sum_{i=1}^{7} i = 28$ arêtes au total) puisqu'ils sont tous partagés par le même élément. On retrouve plusieurs algorithmes plus ou moins sophistiqués dans la littérature pour colorier un graphe, mais l'on se satisfera ici d'un algorithme glouton, parce qu'il est simple à implémenter, rapide et relativement efficace.

Une fois le graphe colorié, on rassemble les nœuds en fonction de leur couleur obtenant ainsi les ensembles de nœuds indépendants. Les nœuds faisant partie d'un même ensemble peuvent être déplacés simultanément sans l'aide de primitives de synchronisation tout en étant assurés de toujours produire un maillage valide. Cette procédure a initialement été développée pour exécuter les algorithmes de déplacement de nœuds sur un ordinateur multicœur, mais elle a également été utilisée avec succès dans des environnements distribués et sur des GPU.

Freitag et al. (1999) étudient aussi l'impact de la division du maillage en ensembles de nœuds indépendants sur le temps de calcul théorique. Puisque le maillage est généré à des fins de simulation numérique pour la MEF ou la MVF, le degré maximal D des nœuds du graphe est borné par $\mathscr{O}(\log N / \log\log N)$, où N est le nombre de nœuds. Le degré maximal des nœuds du graphe borne à la fois le nombre de couleurs nécessaires pour le colorier, soit $D + 1$ pour l'algorithme glouton, et le temps maximal t_{max} passé à lisser un nœud. Alors, si l'on suppose qu'on dispose d'un nombre illimité de cœurs, le temps d'exécution espéré de l'algorithme est donné par la borne suivante :

$$\mathscr{O}(\log N / \log\log N) * t_{max} \tag{2.56}$$

2.5.2 Espaces de parallélisation

Ainsi, il est possible de déplacer certains nœuds simultanément tout en garantissant la conformité du maillage final. La distribution des nœuds d'un ensemble de nœuds indépendants constitue le principal axe de parallélisation, que l'on nommera l'axe *multinœud*. Toutefois, en examinant les opérations qui composent les algorithmes de déplacement de nœuds, on découvre deux sous-axes de parallélisation possibles.

Tous les algorithmes se basant sur la qualité des voisinages élémentaires des nœuds déplacés contiennent une boucle pour mesurer la qualité des éléments des voisinages. Ces voisinages élémentaires sont composés de huit hexaèdres pour les maillages structurés et en moyenne de quinze tétraèdres pour les maillages purement tétraédriques. Les éléments des voisinages n'ont cependant pas à être mesurés dans un ordre particulier. C'est pourquoi il est possible de paralléliser l'évaluation des éléments à chaque nouvelle position. De plus, puisque l'évaluation de la qualité requiert des opérations lourdes comme l'échantillonnage de la métrique en plusieurs points, des inversions de matrices et le calcul de normes, le coût de synchronisation des fils devient négligeable sur GPU. On nommera cet axe de parallélisation l'axe *multiélément*.

Une bonne proportion des algorithmes de déplacement de nœuds évaluent la qualité du voisinage élémentaire pour plusieurs endroits pour déterminer la position optimale du nœud déplacé. Lorsqu'un groupe de positions est connu à l'avance (par exemple les six points d'évaluation pour estimer le gradient de α_P dans la descente du gradient), il est possible d'évaluer la qualité du voisinage élémentaire simultanément pour toutes ces positions. On nommera cet axe de parallélisation l'axe *multiposition*.

On notera que les trois axes de parallélisation sont orthogonaux. En effet, il est possible de composer un espace de parallélisation en choisissant de un à trois des axes proposés. L'espace le plus grand consiste à distribuer les nœuds d'un ensemble indépendant sur l'axe *multinœud*, puis à distribuer les positions intermédiaires des nœuds déplacés sur l'axe *multiposition* et finalement à distribuer les éléments des voisinages élémentaires de chaque nœud pour chaque position intermédiaire sur l'axe *multiélément*. Considérant la très grande quantité de fils d'exécution générée par cette division du travail, la nature hautement parallèle des GPU devient de plus en plus attrayante.

2.6 Déplacement de nœuds sur GPU

La GPGPU est un domaine qui a surtout pris de l'ampleur avec l'arrivée des langages de programmation de haut niveau pour le GPU tels qu'OpenCL, CUDA, GLSL et High Level Shader Language (HLSL). Le GPU est maintenant considéré comme une excellente plateforme

d'accélération en traitement de signal, en intelligence artificielle et en analyse numérique (voir Nguyen (2007)). Pouvant héberger plusieurs milliers de cœurs, les GPU offrent un formidable potentiel de parallélisation. De plus, leur démocratisation, notamment grâce à l'industrie du jeu vidéo, en a fait une plateforme de calcul abordable.

Les programmes écrits pour le GPU sont petits ; bien qu'aucune limite ne soit documentée, mon expérience montre qu'ils ne peuvent dépasser quelques dizaines de milliers d'instructions en langage assembleur. Ces programmes doivent donc se limiter à une tâche spécifique. Les librairies graphiques comme *OpenGL* et *Direct3D* comprenaient à l'origine quatre types de programmes : les nuanceurs de sommet, de géométrie, de tesselation et de fragment. Ces programmes forment un pipeline graphique qui discrétise un modèle polygonal 3D sous forme de pixels à l'écran. Le nuanceur de calcul est un nouveau type de programme exécuté par le GPU. Ces nuanceurs ne font pas partie du pipeline graphique et ont l'avantage de pouvoir travailler sur des primitives arbitraires. Dans le cadre de cette recherche, les algorithmes de déplacement de nœuds seront écrits sous forme de nuanceur de calcul et les nœuds du maillage à adapter constitueront les « primitives » à manipuler. On abordera plus en détail le modèle d'exécution parallèle du GPU à la section 2.6.1.

Dans le domaine de la GPGPU, on dira de deux exécutions concurrentes qui évaluent une condition de branche (*if*, *for*, *while*, *switch*) différemment qu'ils « divergent ». Ce terme se distingue complètement de la notion de divergence en analyse numérique, ce dernier désignant la situation où la solution à notre problème numérique ne peut être approchée. La divergence d'exécutions concurrentes impacte directement l'efficacité du GPU à paralléliser un algorithme. Si pour un groupe de fils d'exécution, la condition de branche est évaluée différemment par au moins l'un des fils, les deux chemins de la branche devront être exécutés séquentiellement. Les fils qui n'ont pas à emprunter une certaine branche sont désactivés pendant son exécution. Tous les fils d'exécution du groupe sont réactivés à la fin du branchement conditionnel. Il s'agit d'une limitation architecturale des GPU que l'on abordera plus en détail à la section 2.6.2.

2.6.1 Modèle d'exécution

Avant les années 2000, les GPU étaient conçus pour accélérer le traitement de deux types de primitives : les triangles et les pixels. Un pipeline graphique doit être mis en place sur le GPU afin d'afficher un modèle géométrique à l'écran. Lorsqu'on travaille avec la librairie *OpenGL*, ce pipeline est formé au minimum d'un nuanceur de sommet et d'un nuanceur de fragment. Les nuanceurs sont de petits programmes exécutés sur le GPU pour traiter en parallèle les primitives qui lui sont envoyées. D'abord, le nuanceur de sommet s'occupe d'appliquer des

transformations affines afin de positionner correctement le modèle géométrique à l'écran en fonction de la position du modèle (*model matrix*), de la position de la caméra (*view matrix*) et de la projection utilisée (*projection matrix*). Le modèle est ensuite discrétisé en fragments selon les pixels qu'il couvre à l'écran. Finalement, le nuanceur de fragment s'occupe de calculer la quantité de lumière réfléchie vers l'observateur pour chacun des fragments.

On remarque trois caractéristiques importantes des nuanceurs. D'abord, ils traitent un très grand volume de données. Un modèle géométrique peut être arbitrairement complexe. Il peut posséder plus d'un million de triangles et demander le traitement de centaines de milliers de fragments, et ce plus d'une dizaine de fois par secondes. Deuxièmement, on remarque que le traitement des primitives est uniforme. Pour les nuanceurs de sommet, les mêmes matrices de transformations affines sont appliquées à tous les sommets. Pour les nuanceurs de fragment, les mêmes textures sont échantillonnées et les mêmes calculs de réflexion de lumières sont effectués. Finalement, il est important de remarquer que le traitement des primitives se fait de manière indépendante. C'est-à-dire qu'un nuanceur de sommet n'a pas besoin de savoir que le modèle est formé de plus d'un triangle pour pouvoir positionner chacun d'eux à l'écran. Il en va de même pour les fragments, qui sont tous parfaitement indépendants.

Ce sont ces caractéristiques algorithmiques qui ont justifié l'évolution des GPU telle qu'on la connaît aujourd'hui. Bien que la venue des langages de programmation de haut niveau ait facilité la progression du calcul généralisé sur GPU et motivé l'unification des unités de traitement, on admettra que les algorithmes qui profitent de la plus grande accélération exhibent toujours ces trois caractéristiques algorithmiques. Cette contrainte vient du modèle d'exécution SIMD des processeurs GPU, c'est-à-dire qu'une instruction est appliquée uniformément à un ensemble de données. Ce modèle contraste avec le modèle d'exécution MIMD des processeurs CPU multicœurs, où plusieurs instructions sont appliquées à plusieurs données. Dans ce cas, pour qu'un algorithme soit parallélisé efficacement, il n'a pas besoin de traiter une grande quantité de données ni de les traiter uniformément et peut admettre une certaine dépendance entre ses exécutions parallèles.

En fait, les GPU s'accordent si bien à l'exécution des nuanceurs de sommet et de fragment que les trois caractéristiques sont devenues des conditions nécessaires qu'un algorithme doit respecter pour être parallélisable sur ce type de processeur. On peut dire que les algorithmes de déplacement de nœuds répondent très bien à la première condition, puisqu'en mécaniques des fluides numérique, l'utilisation de maillages comportant plusieurs millions de nœuds est aujourd'hui la norme. Pour ce qui est des dépendances, on sait que la qualité du voisinage élémentaire associé à un nœud dépend de la position de tous les nœuds voisins. Or, l'utilisation d'ensembles de nœuds indépendants nous garantit que, de tous les nœuds faisant

partie du voisinage élémentaire, seul le nœud central peut bouger. Donc, l'utilisation des ensembles indépendants de nœuds élimine les dépendances entre les exécutions concurrentes de l'algorithme. Notre troisième condition algorithmique est donc vérifiée.

On verra dans les chapitres suivants que la deuxième condition nous causera quelques problèmes, car certaines portions des algorithmes de déplacement de nœuds dans un espace riemannien engendrent des exécutions non uniformes. La première cause est que certains algorithmes, comme la *descente du gradient* et *Nelder-Mead*, ne terminent pas en un nombre fini d'instructions. Ils termineront seulement s'ils ont, mathématiquement parlant, convergé ou s'ils ont atteint un certain nombre d'itérations locales. La deuxième cause vient de l'échantillonnage de la métrique spécifiée. Celle-ci a un impact beaucoup plus grand puisque l'échantillonnage est utilisé par tous les algorithmes et n'est donc pas évitable. L'échantillonnage de la métrique spécifiée constitue l'opération la plus coûteuse et la plus fréquente ; la moindre divergence engendrée par cette opération a des conséquences désastreuses sur l'ensemble du processus d'adaptation.

2.6.2 Architecture matérielle du GPU

On utilisera la nomenclature d'NVIDIA® dans ce document pour toute référence à l'architecture des GPU, parce que sa documentation est plus exhaustive et mieux diffusée que celles d'ATI® et d'Intel®. On commencera par décrire la structure des programmes, puis on expliquera quelles composantes matérielles permettent de paralléliser leur exécution.

On appelle noyau (*kernel*) un programme destiné à être exécuté sur le GPU. Ces noyaux sont l'équivalent des nuanceurs de calcul dans la nomenclature OpenGL. Le corps d'un noyau correspond au corps des boucles imbriquées exécutées séquentiellement que l'on voudrait paralléliser. Par exemple, si l'on veut paralléliser l'exécution de deux boucles *for* imbriquées, ayant respectivement pour compteurs d'incréments i et j, on isolera le corps des boucles dans un noyau et l'on invoquera plusieurs instances de ce noyau sous la forme d'une grille de taille $i_{max} \times j_{max}$. Cette traduction est illustrée à la figure 2.16 sous la forme d'une addition matricielle. La grille de calcul est une représentation conceptuelle en trois dimensions de la disposition des fils d'exécution. Cette grille possède deux niveaux hiérarchiques. D'abord, les fils sont regroupés en blocs de tailles identiques. Puis, ces blocs sont groupés à leur tour pour former une grille de calcul. Il est possible de synchroniser les fils d'exécution appartenant à un même bloc à l'aide de barrières. Cependant, il est impossible de synchroniser les blocs entre eux. Les blocs peuvent tout aussi bien s'exécuter séquentiellement que parallèlement.

Outre l'impossibilité de synchroniser l'exécution de blocs, d'autres aspects doivent être pris en compte. D'abord, la taille des blocs est limitée à 1024x1024x64 fils et le nombre total

```
void matrixSum(
      int row, int col,
      const double* A,
      const double* B,
      double* C)
{
    for(int r = 0; r < row; ++r) {
        for(int c = 0; c < col; ++c) {
            int idx = (r*col) + c;
            C[idx] = A[idx] + B[idx];
        }
    }
}
```

(a) Boucles C++

```
void matrixSum(
      int row, int col,
      const double* A,
      const double* B,
      double* C)
{
    int c = threadIdx.x;
    int r = threadIdx.y;

    int idx = (r*col) + c;

    C[idx] = A[idx] + B[idx];
}
```

(b) Noyau CUDA

Figure 2.16 Traduction de boucles *for* imbriquées en un noyau CUDA

de fils par bloc est limité à 1024. C'est-à-dire que des blocs de tailles 1024x1x1 ou 512x2x1 sont valides, mais qu'une autre de 1024x1024x1 ne l'est pas puisqu'elle dépasse le nombre maximal de fils total par bloc. En plus de permettre la synchronisation des fils d'exécution, l'avantage de regrouper des fils d'exécution en blocs est d'utiliser la mémoire partagée de très faible latence. On utilise généralement la mémoire partagée sous forme de cache gérée manuellement en chargeant de la mémoire globale une petite quantité de données qui sera utilisée fréquemment par les fils d'exécution d'un bloc.

Pour faire la différence avec les processeurs CPU multicœurs (*multi-cores*), on dit des GPU qu'ils sont des processeurs à plusieurs cœurs (*many-cores*). Puisque les GPU peuvent contenir plusieurs milliers de cœurs, ceux-ci sont organisés de manière hiérarchique, un peu à l'image de la grille de calcul. Les cœurs, aussi appelés *Streaming Processors*, sont regroupés en *Streaming Multiprocessors*. La puce d'un GPU est composée de plusieurs *Streaming Multiprocessors*. C'est le *Streaming Multiprocessor* qui administre l'exécution de ses cœurs. Il détient l'unique pointeur d'instruction ainsi que la mémoire partagée et les caches. La carte graphique utilisée pour conduire la présente recherche est une GeForce® GTX 780 Ti d'NVIDIA®. Sa puce, de génération Kepler, contient 15 *Streaming Multiprocessors* contenant chacun 192 *Streaming Processors* ce qui donne un total de 2880 cœurs. Une vue globale de son architecture est donnée à figure 2.17 et un gros plan d'un *Streaming Multiprocessor* est donné à la figure 2.18. La correspondance entre une grille de calcul et la disposition des cœurs du GPU est presque

directe. Un fil d'exécution est assigné à un unique cœur, sans possibilité d'être migré en cours d'exécution. Un bloc est assigné à un unique *Streaming Multiprocessor*, sans possibilité d'être migré non plus. Cependant, un bloc peut contenir plus de fils d'exécution que le *Streaming Multiprocessor* ne possède de cœurs. Un bloc est toujours divisé en *warps*. Pour minimiser les temps d'attente causés par la latence des différentes mémoires, le *Streaming Multiprocessor* alternera entre ses différents *warps* pour maximiser son taux d'occupation. Un *Streaming Multiprocessor* peut non seulement exécuter simultanément plusieurs *warps* d'un même bloc, mais aussi plusieurs blocs en même temps.

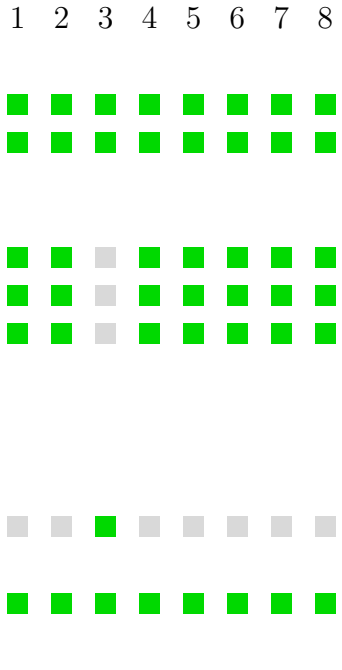
En faisant la lumière sur cette architecture, on comprend mieux pourquoi la divergence a un si grand impact sur l'exécution d'un noyau. Les cœurs n'ont pratiquement aucune autonomie. Il n'y a qu'un seul pointeur d'instruction par *warp*. À la rencontre d'une condition de branche, si tous les fils d'exécution du *warp* évaluent cette condition à vrai ou faux, seule la branche activée sera exécutée par le *warp*. Par contre, si au moins un fil évalue la condition différemment, les deux branches seront exécutées séquentiellement. Les cœurs possèdent un

Figure 2.17 La puce de la GeForce GTX 780 Ti contient 15 *Streaming Multiprocessors* (SMX) [2]

2. Tirée de NVIDIA (2012)

Figure 2.18 Un *Streaming Multiprocessors* de la GeForce 780 Ti contient 192 unités en simple précision (Core), 64 unités en double précision (DP Unit), 32 unités *load/store* (LD/ST) et 32 unités de fonctions spéciales (SFU), tirée de NVIDIA (2012)

masque d'activité. Si ce masque est désactivé, il remplacera les instructions par des *noop*s. Par exemple, prenons le cas d'un *warp* de 8 fils d'exécution où, à la rencontre d'une branche *if-else*, un des fils évalue la condition à vrai tandis que les 7 autres évaluent la condition à faux. Le *Streaming Multiprocessor* chargera les instructions de la branche *if*, et 7 des fils désactiveront leur masque d'exécution. Seul le fil qui a évalué la condition à vrai exécutera réellement les instructions. À la rencontre de la branche *else*, les 7 fils inactifs réactiveront leur masque tandis que l'autre fil désactivera son masque, puis le *Streaming Multiprocessor* chargera les instructions de la branche *else*. Une fois sortie de la structure de contrôle, les 8 fils du *warp* activeront leur masque avant que le *Streaming Multiprocessor* ne charge les prochaines instructions du noyau. On dira à ce point que les fils ont convergé. La figure 2.19 montre l'état du masque d'activité à chaque instruction d'un noyau comportant une structure de contrôle *if-else*.




```
vec3 qualityToColor(in float quality)
{
    vec3 color = vec3(0, 0, 0);
    if(quality >= 0.0)
    {
        // Red to green
        color.r = 1.0 - quality;
        color.g = quality;
        color.b = 0.0;
    }
    else
    {
        // White
        color = vec3(1.0, 1.0, 1.0);
    }
    return color;
}
```

Figure 2.19 Les cœurs d'un même *warp* utilisent un masque d'activation pour n'exécuter que les instructions de la branche choisie

2.7 État de l'art en déplacement de nœuds sur GPU

Les articles présentés ici concernent tous le développement d'algorithmes de déplacement de nœuds sur GPU. La plupart se concentrent uniquement sur l'optimisation de maillage (c.-à-d. sans fonction métrique). L'un d'entre eux intègre le déplacement de nœuds à un cadre plus large d'optimisation, incluant également des opérations topologiques exécutées en parallèle sur le CPU. Une autre aborde l'adaptation de maillage dans son ensemble en tenant compte d'une fonction métrique. Toutefois, on constatera qu'il reste encore beaucoup de travail à accomplir avant d'arriver à un logiciel complet d'adaptation de maillages tridimensionnels formés d'éléments non simpliciaux dont le déplacement de nœuds est effectué sur le GPU.

Rokos et al. (2011) présentent une première solution pour paralléliser l'adaptation de maillage par déplacement de nœuds sur GPU. Écrit en CUDA, leur logiciel permet d'adapter des maillages 2D composés uniquement d'éléments simpliciaux. La seule technique de déplacement de nœuds implémentée par les auteurs est une version modifiée du lissage laplacien. L'aspect le plus intéressant de leur recherche est l'utilisation d'une texture pour stocker la métrique spécifiée. Bien qu'on ne connaisse pas la technique alternative qu'ils utilisent pour échantillonner la métrique, on peut supposer qu'il s'agit d'un maillage de fond non structuré. Les auteurs affirment que l'utilisation d'une texture est l'optimisation qui a permis le plus grand gain en performance. Dans le vocabulaire infographique, une texture est une image stockée en mémoire où chaque pixel est représenté par un quadruplet Rouge-Vert-Bleu-Alpha. La forme classique du quadruplet est donnée sur 32 bits et chaque composante est représentée par un entier sur 8 bits : Red8I_Green8I_Blue8I_Alpha8I. Les GPU récents sont capables de stocker les pixels sous forme de nombres à virgule flottante de 32 bits par composante : Red32F_Green32F_Blue32F_Alpha32F. Dans le cas 2D, cette représentation permet de stocker les trois composantes du tenseur métrique spécifié dans un pixel. Il suffit de choisir une résolution appropriée à la taille de notre problème de mécanique des fluides et de discrétiser la métrique spécifiée dans la texture. L'échantillonnage de texture est une opération fondamentale sur GPU. L'échantillonnage et l'interpolation de texture bénéficient d'une accélération matérielle, allégeant ainsi les noyaux de calcul. Par contre, l'utilisation d'une texture possède certains inconvénients. Une texture est essentiellement un tableau de valeurs. Contrairement à un maillage de fond non structuré, elle offre peu de contrôle sur la précision spatiale de la métrique. Si l'on veut respecter la précision du maillage de fond original, il faudrait que la taille des pixels soit plus petite ou égale à la taille du plus petit élément du maillage de fond. Malheureusement, ceci pourrait rapidement mener à des textures beaucoup trop volumineuses pour entrer en mémoire. Cependant, les auteurs estiment que la métrique spécifiée n'est qu'un indicateur et n'a pas besoin d'être parfaitement respectée.

On peut donc se permettre un peu d'imprécision dans sa représentation spatiale. De plus, si l'on veut être conservateur, on peut toujours assigner à chaque cellule de la texture le tenseur métrique le plus restrictif que cette cellule couvre. Les auteurs ont réussi à obtenir un déplacement de nœuds jusqu'à 148 fois plus rapide par rapport à leur version séquentielle. À noter qu'aucune mention des qualités initiales et finales des maillages n'est faite. Il est donc impossible d'évaluer la validité ou l'efficacité de leurs algorithmes d'adaptation. Il aurait été très pertinent de connaître l'impact des textures sur la convergence du processus d'adaptation.

D'Amato and Vénere (2013) offrent une solution pour les maillages tridimensionnels, mais seulement composés d'éléments simpliciaux sans métrique riemannienne. Leur solution généralise la manipulation des techniques d'adaptation de maillage par l'intermédiaire de *clusters*. Un *cluster* est l'ensemble des éléments qui seront modifiés lors d'un déplacement de nœuds ou d'une modification topologique. Pour le déplacement de nœuds, le *cluster* correspond exactement au voisinage élémentaire du nœud. Bien que leur logiciel manipule les opérations topologiques et géométriques de manière uniforme, seuls les déplacements de nœud ont l'opportunité d'être exécutés par le GPU. Ce qui rend difficile l'exécution des opérations de modifications topologiques sur le GPU est la génération et la destruction d'éléments et d'arêtes puisque l'allocation et la libération dynamique de mémoire doivent être réalisées manuellement sur des tampons mémoires de tailles fixes. Chaque opération d'adaptation exécutée dans leur logiciel définit une fonction de portée. Cette fonction retourne, pour un nœud donné, l'ensemble des éléments qui seront potentiellement affectés lors de son exécution. Avant d'exécuter les opérations, les nœuds sont ordonnés en fonction de leur qualité. Puis le logiciel construit un ensemble de nœuds indépendant en tentant d'assigner, du pire nœud au meilleur, une opération d'adaptation. Lorsque l'ensemble de nœuds indépendants ne peut être agrandi davantage, les opérations sont exécutées en parallèle et le prochain ensemble de nœuds est construit à partir des nœuds qui n'ont pas encore été traités. Cette conception permettrait de maximiser l'utilisation du CPU et du GPU en simultané. Il peut toutefois être difficile de définir la fonction de portée pour certaines modifications topologiques comme le retournement d'arêtes et le retournement multiface. Les auteurs ont réussi à accélérer le processus complet d'adaptation de maillage (avec modifications topologiques) jusqu'à 34 fois par rapport à leur version séquentielle. À noter que leur solution n'est pas généralisée aux espaces riemanniens. C'est-à-dire qu'ils n'ont pas à échantillonner de tenseurs métriques sur un maillage de fond.

Mei et al. (2014) proposent une autre solution pour paralléliser le déplacement de nœuds de maillages 2D composés d'éléments simpliciaux, encore une fois sans métrique riemannienne. Cet article ajoute peu à celui Rokos et al. (2011), mais soulève un questionnement sur la

manière optimale de stocker le maillage en mémoire sur le GPU. Il existe deux grandes classes de structures de données sur un GPU : la structure de vecteurs et le vecteur de structures. D'une part, on aimerait minimiser l'empreinte du maillage en mémoire, mais de l'autre on voudrait profiter du motif d'accès mémoire qui possède la plus petite latence. Les conclusions de l'article sur la structure de données à utiliser en adaptation de maillage sont ambiguës. Selon leurs expérimentations, le vecteur de structures alignées en mémoire est plus rapide que la structure de vecteurs, qui est, elle, plus rapide que le vecteur de structures non alignées. L'alignement en mémoire est une instruction donnée par le programmeur au compilateur pour que les adresses des éléments d'un vecteur soient toujours un multiple d'un certain nombre d'octets. Aligner les éléments d'un vecteur garantit généralement des accès mémoires plus rapides, au prix d'espaces mémoire perdus. Par exemple, si une structure stocke les coordonnées x, y, et z des nœuds en double précision, et que l'on aligne cette structure de $8+8+8 = 24$ octets sur 32 octets, elle occupera $32-24 = 8$ octets de plus par élément que sa version non alignée ce qui représente une augmentation de 33%. Les résultats expérimentaux de l'article montrent clairement le gain en vitesse offert par le vecteur de structures alignées par rapport aux autres structures de données, mais en regardant de plus près le listage des structures, on s'aperçoit que l'alignement n'est pas la seule différence entre les versions dites « alignée » et « mésalignée » du vecteur de structures. Dans le cas « mésaligné », la structure des nœuds possède un vecteur local de taille fixe (64) pour stocker les index des nœuds voisins. Dans le cas « aligné », tous les vecteurs ont été extraits des nœuds pour être stockés dans un unique vecteur global. Non seulement cette modification dépasse le cadre de l'alignement, mais elle réduira de manière certaine la quantité de mémoire copiée par un algorithme. Dans le lissage laplacien, on accède aux nœuds voisins pour connaître leur position. Ce qui signifie que, dans le cas « mésaligné », on copiera une liste inutilisée de 256 octets chaque fois qu'on voudra connaître la position d'un voisin, ce que l'on évite dans le cas « aligné ». Bref, les résultats de cet article démontrent surtout qu'il est important de ne pas copier inutilement de la mémoire. Les auteurs ont obtenu une version GPU jusqu'à 15 fois plus rapide que sa version séquentielle sur CPU.

Dahal and Newman (2014) présentent d'autres algorithmes de déplacement de nœuds parallélisés pour des maillages 2D composés d'éléments simpliciaux sans métrique riemannienne. Ils ont testé trois algorithmes différents de déplacement de nœuds : le lissage laplacien, le lissage de Zhou et Shimada et le lissage de Xu et Newman. Les auteurs ont observé des accélérations maximales de 40x, 70x et 114x respectivement pour chacun de leurs algorithmes par rapport à leurs versions séquentielles sur CPU.

Cheng et al. (2015) présentent la solution la plus complète à ce jour pour paralléliser les algorithmes de déplacement de nœuds sur le GPU. Leur solution cible les maillages 3D composés

d'éléments simpliciaux et n'offre encore une fois aucun support pour utiliser des métriques riemanniennes. Tout d'abord, on remarquera qu'il s'agit du premier article à mentionner explicitement une mesure de qualité. L'inverse du rapport des moyennes, tel que présenté à la section 2.3.3, est utilisé pour mesurer la qualité des tétraèdres. De plus, il s'agit du seul article à traiter différemment les nœuds frontières. Pour limiter la divergence, plusieurs types d'ensembles de nœuds indépendants sont définis : un pour les nœuds intérieurs, un pour les nœuds de surface et un autre pour les nœuds sur les arêtes de la frontière. Seuls les nœuds intérieurs sont traités par le GPU. Ce traitement hétérogène des nœuds est justifié par le fait que les nœuds frontières doivent toujours être reprojetés sur la frontière à laquelle ils appartiennent, ce qui représenterait une source importante de divergence pour le GPU. On traitera donc les nœuds frontières sur le CPU. Cette manière de distribuer le traitement des nœuds fait émerger un second niveau de parallélisme. En effet, si tous les nœuds sont traités par le GPU, le CPU doit bloquer son exécution en attendant que l'exécution du noyau soit terminée. Pourquoi ne pas occuper le CPU en lui attribuant quelques nœuds pendant que le GPU traite les nœuds intérieurs ? Le développement d'une technique plus sophistiquée d'équilibration des charges de calcul par l'attribution dynamique de nœuds au CPU fait partie des recherches futures des auteurs. L'article est également le seul à se pencher sur le problème des maillages trop volumineux pour entrer en mémoire principale, bien qu'aucune solution ne soit proposée dans cet article. Les auteurs ont obtenu une accélération sur GPU maximale de 16x par rapport à leur version séquentielle sur CPU. On rappellera que la solution présentée ne supporte pas les éléments non simpliciaux ni les métriques riemanniennes.

2.7.1 Lacunes en adaptation sur GPU

Pratiquement tous les objectifs de ma recherche sont abordés dans l'un ou l'autre des articles présentés ici, mais jamais tous à la fois dans un même article. Plusieurs algorithmes de lissage ont été portés sur le GPU avec succès. Ces algorithmes permettent d'optimiser la qualité des maillages triangulaires et tétraédriques beaucoup plus rapidement que les algorithmes parallèles sur CPU. On a même vu qu'il était possible d'introduire une métrique riemannienne dans ces algorithmes pour adapter des maillages bidimensionnels.

Il reste toutefois certains objectifs à atteindre afin de pouvoir tirer profit de cette accélération pour l'adaptation de cas réels en mécanique des fluides numériques. Il serait d'abord souhaitable de pouvoir traiter des maillages 3D composés d'éléments non simpliciaux comme les pyramides, les prismes et les hexaèdres puisqu'ils sont fréquemment rencontrés dans la pratique. De plus, on remarque que d'autres espaces de parallélisation auraient intérêt à être explorés afin de réduire davantage le temps de calcul. On voudrait également étendre l'utili-

sation de métrique riemannienne à la troisième dimension. Finalement, on voudrait connaître l'impact d'une nouvelle représentation de la métrique spécifiée. Par exemple, stocker et échantillonner la métrique spécifiée sous forme de texture semble accélérer grandement l'exécution sur GPU, mais est-ce aux dépens de la qualité finale du maillage ?

CHAPITRE 3 INFRASTRUCTURE CPU

Les deux chapitres qui suivent décrivent l'architecture du logiciel d'adaptation de maillage développé dans le cadre de ce projet de maîtrise. Le présent chapitre se concentre sur l'infrastructure logicielle du côté CPU. On y décrit l'implémentation de plusieurs algorithmes de modifications topologiques et de déplacement de nœuds qui constituent l'état de l'art actuellement en adaptation de maillage. Les algorithmes de déplacement de nœuds développés pour le CPU peuvent être exécutés séquentiellement et parallèlement. Les versions séquentielles sont les points de référence dans l'évaluation des algorithmes parallèles. Le chapitre qui suit se concentrera quant à lui sur l'infrastructure logicielle du côté GPU, soit les implémentations CUDA et GLSL des mêmes algorithmes de déplacement de nœuds et leur intégration au logiciel d'adaptation. On abordera également, à ce chapitre, le problème de conception de structures de données appropriées pour le GPU.

L'adaptateur de maillage développé pour ce projet a été conçu comme un laboratoire de tests. Les modules présentés dans ce chapitre possèdent différentes instances qui peuvent être échangées à l'exécution. De cette manière, des rapports de comparaison entre différents types d'algorithmes (ex. : Descente du gradient vs Nelder-Mead) et différentes versions d'algorithmes (ex. : C++ séquentiel vs C++ parallèle vs GLSL vs CUDA) peuvent être produits en lot. On présentera d'abord l'architecture logicielle de l'adaptateur et l'on terminera par la description des structures de données manipulées par les algorithmes sur le CPU.

3.1 Architecture de l'adaptateur

L'adaptateur comporte cinq modules : l'échantillonneur, le mesureur, l'évaluateur, le lisseur et le topologiste. Le mesureur a la particularité de ne pas pouvoir être choisi par l'utilisateur. L'instance utilisée dépend du type d'échantillonneur choisi, c'est-à-dire du type de métrique demandée par l'utilisateur. Les modules forment une chaîne où un module donné produit ses résultats en fonction des modules qui le précèdent dans cette chaîne. Le diagramme de paquetages de l'adaptateur est donné à la figure 3.1. L'échantillonneur a pour tâche d'échantillonner la métrique spécifiée. Le mesureur utilise les échantillons de métrique pour mesurer les dimensions des arêtes, des faces et des éléments dans l'espace riemannien spécifié. L'évaluateur utilise à son tour ces dimensions pour évaluer la qualité des éléments du maillage. Finalement, le lisseur détermine la position optimale des nœuds en fonction de la qualité des voisinages élémentaires. De la même manière, le topologiste apporte des modifications topologiques au maillage en fonction de la qualité des éléments. Dans l'architecture actuelle,

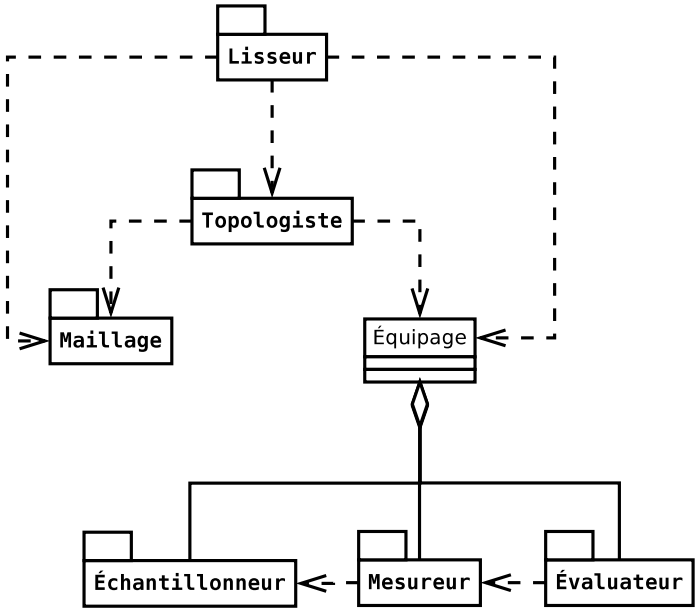


Figure 3.1 Diagramme de paquetages du logiciel d'adaptation

le lisseur est le seul module en mesure de déterminer les bons moments pour invoquer le topologiste, car il gère également l'exécution des noyaux de calcul sur le GPU. C'est pourquoi le topologiste est en quelque sorte l'« esclave » du lisseur. Les modules d'échantillonnage, de mesure et d'évaluation sont nécessaires à toutes les opérations d'adaptation. C'est pourquoi ils sont regroupés dans une petite classe nommée Équipage (voir figure 3.1).

3.1.1 Le maillage

Le maillage est la structure de données centrale de l'adaptateur. Il contient à la fois la représentation des frontières, telle que fournie par un logiciel de Conception Assistée par Ordinateur (CAO), et les tableaux de nœuds, d'éléments et de dictionnaires topologiques.

La frontière est un graphe tridimensionnel composé de sommets, d'arêtes et de faces. Les arêtes et les faces peuvent être linéaires ou courbes. Elles peuvent être définies à l'aide de fonctions analytiques, de courbes de Bézier ou de NURBS. La frontière indique au maillage quelle est la portion de l'espace qui doit être remplie d'éléments.

Les données fondamentales constituant le maillage sont les tableaux de nœuds et d'éléments. Les références entre les différents tableaux se font à l'aide d'indices. C'est-à-dire qu'un élément ne fait pas référence à ses nœuds à l'aide de pointeurs en mémoire, mais par leurs indices dans le tableau de nœuds. On verra au chapitre 4 que cette manière de procéder facilitera la conception des structures de données sur GPU.

3.1.2 Les échantillonneurs

La carte de métriques spécifiées est fournie en entrée du processus d'adaptation. La manière standard de représenter cette carte consiste à stocker les tenseurs métriques aux nœuds du maillage initial. On obtient ainsi un maillage de fond sur lequel il est possible d'échantillonner la métrique en interpolant les tenseurs sur ses éléments. Certaines structures de données et algorithmes doivent être mis en place pour accélérer l'échantillonnage de la métrique. Les échantillonneurs ont pour tâche d'encapsuler cette complexité et d'offrir au reste du système une interface simple pour évaluer la fonction métrique en un point quelconque de l'espace.

Plusieurs approches d'échantillonnage ont été testées dans le cadre de cette recherche. Quelques-unes ont un intérêt strictement théorique, d'autres plus pratique ; certaines offrent des performances encourageantes sur CPU et GPU, d'autres sont impraticables sur le GPU. L'échantillonnage de la métrique est une opération déterminante pour les adaptateurs. De tous les modules, il s'agit du plus fréquemment invoqué et du plus gourmand en temps de calcul. Bref, comme on le constatera au chapitre 5, c'est le module qui a le plus grand impact sur les performances de l'adaptateur sans compter qu'il influence de manière importante la qualité du maillage adapté. C'est pourquoi plusieurs approches pour échantillonner la métrique spécifiée, qui sont décrites ici, ont été testées. On garde la présentation de l'échantillonnage par texture pour le chapitre suivant, car son implémentation est intimement liée au GPU.

Uniforme

L'échantillonneur uniforme est le plus simple et le plus limité de tous. Il ne supporte qu'un facteur d'échelle isotrope et constant sur l'ensemble du domaine. Cet échantillonneur n'a qu'un intérêt théorique puisqu'il ne permet pas de spécifier des tailles locales, ce qui est essentiel en adaptation de maillage. Il est utilisé principalement pour déboguer le reste du système. De plus, lorsqu'on tente d'évaluer les performances relatives entre les différentes versions des algorithmes de déplacement de nœuds (C++ séquentiel, C++ parallèle, GLSL et CUDA), cet échantillonneur permet d'isoler l'impact de la métrique et par conséquent de mieux apprécier l'impact des autres modules.

Analytique

Contrairement à l'échantillonneur uniforme, l'échantillonneur analytique permet de spécifier un champ de tenseurs métriques anisotropes qui varie dans l'espace. Par contre, puisque cette implémentation échantillonne la métrique à partir d'une fonction analytique et non à partir du maillage de fond, elle offre elle aussi un intérêt purement théorique. En plus de permettre

le débogage du système et de réduire l'impact de l'échantillonnage sur les performances, elle permet de tester la robustesse des algorithmes d'adaptation. Notez que plusieurs cas de test standards en adaptation de maillage utilisent des métriques analytiques. L'échantillonneur analytique permet donc de vérifier le processus d'adaptation en reproduisant ces cas de test.

Recherche locale

On appellera « recherche locale » la technique classique qui permet d'interpoler la métrique spécifiée directement sur le maillage de fond. Le terme « recherche locale » fait référence au fait que l'on tente de trouver l'élément hébergeant l'échantillon demandé en se déplaçant d'éléments voisins en éléments voisins. C'est-à-dire que pour toutes requêtes d'échantillon, une estimation initiale doit être fournie. À partir de cette estimation, on se déplacera vers le voisin qui a le plus grand potentiel de contenir l'échantillon recherché. Ce type de recherche demande une certaine préparation. Il est préférable d'abord de travailler sur un maillage purement tétraédrique. Dompierre et al. (1999) proposent une technique robuste pour subdiviser les prismes, pyramides et hexaèdres en tétraèdres en se basant sur les indices des nœuds. Cette technique assure la génération d'un maillage tétraédrique conforme. Il faut ensuite identifier le voisinage des éléments, c.-à-d. déterminer vers quel tétraèdre on se dirige si l'on traverse une face donnée d'un élément. Puisque les tétraèdres reposant sur la frontière ont des faces qui n'ont pas de voisin, on utilisera un marqueur particulier pour indiquer que ces faces ne sont pas traversables. La figure 3.2 est un exemple 2D du voisinage d'un triangle sur une frontière.

À partir d'une estimation initiale, il n'est pas toujours évident de savoir quel voisin emprunter pour atteindre l'échantillon demandé. Frey and George (1999) utilisent le système de coordonnées barycentriques comme heuristique. Si M est la position de l'échantillon, celle-ci peut être exprimée comme un barycentre (ou combinaison linéaire) du tétraèdre de sommets

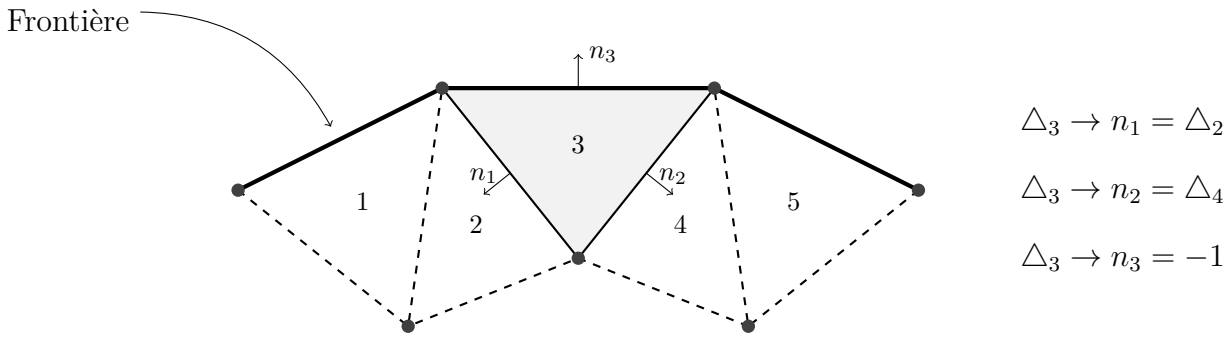


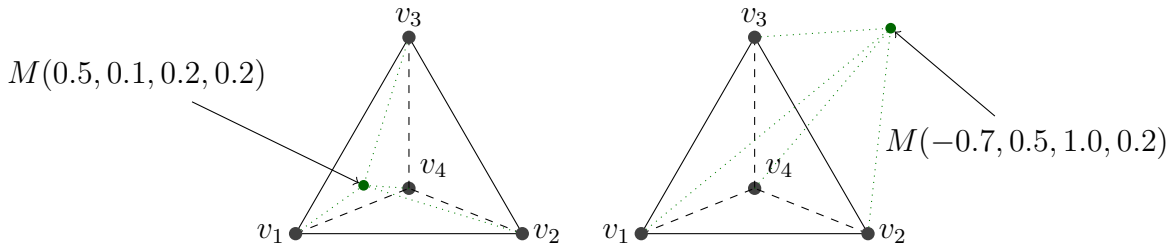
Figure 3.2 Étiquetage des faces en fonction des éléments voisins

v_i. Les coordonnées barycentriques sont les coefficients λ_i donnés à l'équation 3.1. Calculer les coordonnées barycentriques de l'échantillon en fonction du tétraèdre courant peut nous mener vers deux situations : soit toutes les coordonnées sont positives et alors l'échantillon est à l'intérieur même de l'élément courant, soit au moins une des coordonnées est négative et la plus petite des coordonnées négatives est un bon indicateur de la face à traverser pour atteindre l'échantillon (voir figure 3.3). L'avantage de cette méthode est que les coordonnées barycentriques servent à la fois à déterminer si l'échantillon est contenu dans le tétraèdre courant et, dans le cas contraire, à déterminer quelle face traverser pour s'en approcher.

$$M(\lambda_1, \lambda_2, \lambda_3, \lambda_4) = \lambda_1 v_1 + \lambda_2 v_2 + \lambda_3 v_3 + \lambda_4 v_4, \text{ avec } \sum_{i=1}^{4} \lambda_i = 1 \tag{3.1}$$

Malheureusement, l'utilisation des coordonnées barycentriques n'est pas une heuristique robuste. Cette méthode mène parfois à tourner en rond dans un maillage tétraédrique. Une solution possible consiste à ajouter quelques techniques algorithmiques comme une liste d'éléments tabous ou à traverser une face aléatoire de temps à autre pour échapper aux cas problématiques, mais cela n'offre toujours aucune garantie. Il existe cependant des techniques plus robustes pour s'approcher de l'échantillon. Le logiciel d'adaptation développé utilise une variante du lancer de rayon. On place d'abord l'origine du rayon au centre de l'élément estimé et on le fait pointer vers la position de l'échantillon désiré. Puisque, par définition, un tétraèdre est l'enveloppe convexe d'un ensemble de quatre points, on sait que si un rayon entre, il en sortira par un et un seul point. De plus, puisque l'on suit une droite, il n'est plus possible de tourner en rond cette fois. La variante 2D de cette recherche par suivi de rayon est illustrée à la figure 3.4.

Contrairement aux autres échantillonneurs, la recherche locale ne peut pas trouver un échantillon efficacement en se basant uniquement sur la position de l'échantillon. Sans estimation



(a) Intérieur (coordonnés positives) (b) Extérieur (au moin une négative)

Figure 3.3 Coordonnées barycentriques d'un échantillon en fonction de sa position relative à un tétraèdre

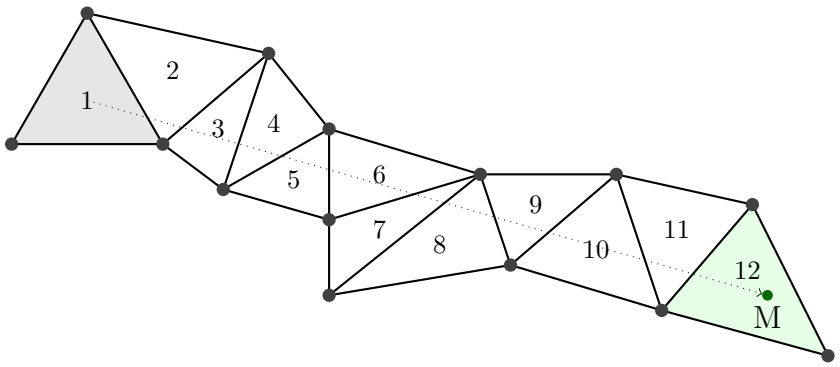


Figure 3.4 Le maillage est traversé en ligne droite de l'élément initial vers l'échantillon

initiale, sa complexité algorithmique est $\mathcal{O}(n^{1/3})$, où n est le nombre d'éléments dans le maillage de fond. Cette complexité vient du fait que le volume V du maillage est proportionnel au cube de son rayon r. De plus, si l'on amorçait la recherche d'une position arbitraire, on traverserait en moyenne l'équivalent du rayon du maillage pour atteindre chaque échantillon. Au contraire, avec une estimation initiale, on remarque en pratique que très peu d'éléments doivent être traversés pour atteindre les échantillons, réduisant ainsi la complexité de l'algorithme à $\mathcal{O}(1)$. Ces estimations doivent être gardées en mémoire et mises à jour chaque fois qu'elles sont utilisées pour échantillonner la métrique spécifiée. On peut soit stocker ces estimations aux nœuds ou aux éléments du maillage. Pour éviter une mise à jour trop fréquente, l'adaptateur conserve une estimation par nœud et une estimation par coin d'élément. Cela fait une estimation par nœud, un par tétraèdre, quatre par pyramide, six par prisme et huit par hexaèdre. Les estimations aux nœuds sont utilisées pour calculer la longueur des arêtes et les estimations aux coins des éléments servent aux mesures de la qualité.

3.1.3 Les mesureurs

Les mesureurs sont des spécialisations pour mesurer les longueurs, les aires et les volumes en fonction du type de métrique spécifiée choisie. Deux mesureurs ont été implémentés afin d'optimiser les calculs en fonction du choix du type de métrique.

Indépendant de la métrique

Le mesureur indépendant de la métrique fait tous ses calculs dans l'espace euclidien, tout en respectant le facteur de mise à l'échelle. L'avantage de ce mesureur est qu'il est simple et rapide. Ce mesureur est utilisé principalement pour déboguer les autres modules et pour éliminer tout coût de calcul associé à la métrique. Le seul échantillonneur à requérir ce

mesureur est l'échantillonneur uniforme.

Dépendant de la métrique

Le mesureur dépendant de la métrique est la généralisation aux espaces riemanniens du mesureur indépendant de la métrique. On se rappellera que les calculs de longueurs, d'airs et de volumes sont définis à l'aide d'intégrales. Ce mesureur s'occupe d'approximer numériquement le calcul des longueurs selon un seuil de tolérance. Un calcul adaptatif permet de mesurer la longueur d'une arête en subdivisant l'arête en sous-segments là où la métrique varie le plus rapidement. La longueur totale de l'arête est donnée par la somme des longueurs de ses segments mesurés dans la métrique spécifiée. La figure 3.5 illustre la subdivision du calcul de longueur pour une métrique donnée. Les calculs d'airs et de volumes sont quant à eux approximés à l'aide de quadratures de Gauss.

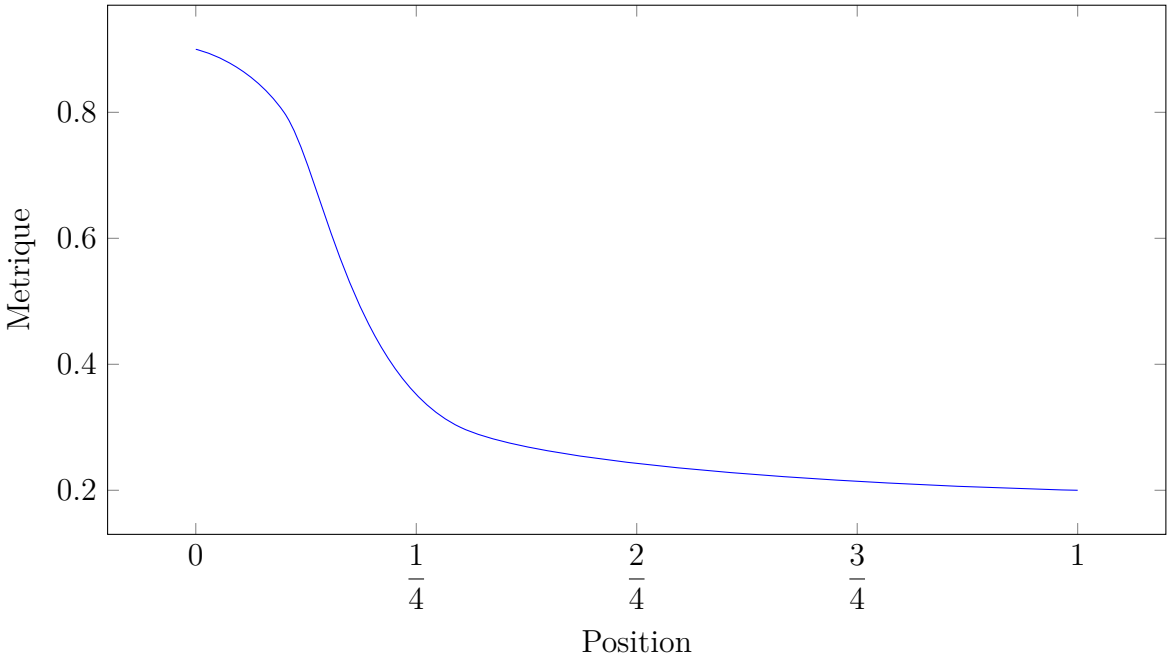
3.1.4 Les évaluateurs

De toutes les mesures de qualité présentées à la section 2.3, seules celles pouvant s'étendre aux éléments non simpliciaux ont été retenues. Les deux évaluateurs développés s'utilisent dans des contextes différents. Le premier évaluateur est destiné à l'optimisation pure de maillage, sans carte de spécification pour la taille et la forme des éléments. On l'utilisera en conjonction avec l'échantillonneur uniforme pour donner une idée des performances de l'adaptateur sans métrique riemannienne. Le deuxième évaluateur mesure la qualité des éléments en fonction de la métrique spécifiée. C'est l'évaluateur à utiliser en adaptation de maillage.

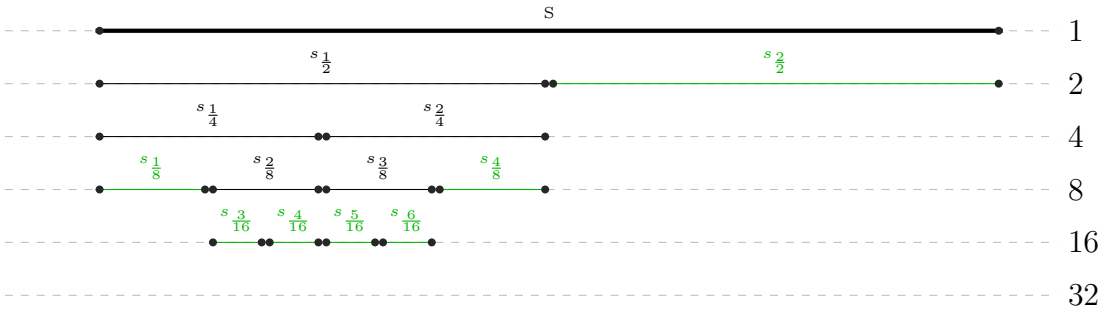
Les évaluateurs offrent deux méthodes pour calculer la qualité d'un ensemble d'éléments. D'abord, ils permettent de calculer la qualité d'un voisinage élémentaire, peu importe si le type de ses éléments est homogène ou non. Cette méthode est fondamentale pour tous les algorithmes de déplacement de nœuds. Les évaluateurs permettent également de calculer la qualité d'un maillage dans son ensemble en donnant la qualité du pire élément et la moyenne harmonique des qualités de tous les éléments. Cette méthode est utile pour vérifier la convergence du processus d'adaptation et pour comparer les gains de qualité entre les algorithmes de déplacement de nœuds.

Rapport des moyennes

En optimisation de maillage sans métrique riemannienne, on utilisera le rapport des moyennes dans sa version algébrique, η (équation 2.27), puisqu'elle est la seule mesure euclidienne à supporter les éléments non simpliciaux. Cette mesure est sensible à la forme des éléments et



(a) Fonction métrique



$$l_M(S) = l_M(s_{\frac{1}{8}}) + l_M(s_{\frac{3}{16}}) + l_M(s_{\frac{4}{16}}) + l_M(s_{\frac{5}{16}}) + l_M(s_{\frac{6}{16}}) + l_M(s_{\frac{4}{8}}) + l_M(s_{\frac{2}{2}})$$

(b) Subdivision de l'arête

Figure 3.5 Calcul adaptatif de la longueur d'une arête dans la métrique

non à leur taille. Une unique fonction permet de calculer le rapport des moyennes à partir de deux représentations matricielles : l'une associée au tétraèdre de référence et l'autre associé au coin d'un élément. Cette fonction est directement utilisée pour mesurer la qualité des tétraèdres du maillage. Pour les pyramides, les prismes et les hexaèdres, la fonction doit être appelée pour chacun des coins de l'élément. La qualité d'un élément non simplicial est donnée par la moyenne harmonique des qualités de ses coins (équations 2.39 à 2.42). Pour la pyramide et le prisme, il faut faire attention à l'ordre des colonnes des représentations matricielles des tétraèdres de référence F_k. Bien qu'il existe six manières d'assembler les matrices, une seule est valide. Cette configuration dépend de l'assemblage de la matrice de référence F_r choisie (voir figure 2.5). Ce problème ne survient pas avec l'hexaèdre puisque sa matrice F_r est invariablement la matrice identité.

Conformité à la métrique

Pour l'adaptation de maillage avec métrique riemannienne, on utilisera la mesure de conformité à la métrique $\mathscr{C}$. Son implémentation est très similaire à celle du rapport des moyennes. Une première fonction permet d'approximer la métrique spécifiée fournie par le maillage de fond à l'aide d'une quadrature de Gauss. Une deuxième fonction mesure la qualité du même tétraèdre en fonction de la métrique spécifiée M_S et de la métrique de l'élément M_K. La qualité des tétraèdres du maillage est donnée directement par la deuxième fonction tandis que la qualité des éléments non simpliciaux doit être calculée par la moyenne harmonique des coins (équations 2.39 à 2.42). Tout comme pour le rapport des moyennes, l'ordre des colonnes des représentations matricielles des tétraèdres de référence F_k doit correspondre à la forme des matrices F_r choisie parmi les six permutations de colonnes possibles (voir figure 2.5) sans quoi la mesure sera invalide.

3.1.5 Les lisseurs

Le module de lissage est le cœur du système d'adaptation. À l'aide des modules d'échantillonnage, de mesure et d'évaluation, les lisseurs déplacent itérativement les nœuds du maillage jusqu'à ce que le processus converge, c'est-à-dire jusqu'à ce qu'il ne soit plus possible d'améliorer la qualité du maillage sans avoir recours à des modifications topologiques. Les algorithmes implémentés dans le logiciel d'adaptation ont été choisis pour mettre en lumière les forces et les faiblesses du parallélisme sur GPU.

Laplacien à ressorts

Bien que le lissage laplacien dans sa forme classique (ressorts de traction) ait tendance à produire des éléments dégénérés, surtout près des concavités, il a été retenu pour sa simplicité. L'algorithme est facile à implémenter, autant sur le CPU que sur le GPU, ne nécessite pas de primitives de synchronisation et est l'algorithme le plus rapide de tous ceux testés dans ce projet. De plus, le lissage laplacien est l'algorithme le plus populaire parmi les articles cités en déplacement de nœuds sur GPU. Il offre ainsi une bonne base de comparaison avec les recherches actuelles.

Le lissage laplacien à ressorts s'exécute en quatre étapes, la dernière étant particulière aux nœuds frontières. Pour un nœud donné, on calcule la position du centre de son voisinage élémentaire. Puis, on détermine l'endroit où sera déplacé le nœud par interpolation linéaire entre la position actuelle du nœud et le centre de son voisinage élémentaire. Un coefficient de « frottement » spécifie si la destination choisie est plus proche de la position d'origine ou du centre. Ce coefficient ralentit artificiellement la convergence de l'algorithme, mais offre de la stabilité. De plus, étant donné que le lissage laplacien est un processus itératif, le centre actuel du voisinage élémentaire n'est pas la position optimale du nœud dans le maillage final, puisque le centre se déplace au fur et à mesure que les nœuds voisins sont déplacés. Une fois la destination déterminée, il ne reste plus qu'à déplacer le nœud à cet endroit. Si le nœud repose sur une frontière, il faut également prendre soin de le projeter sur son support topologique, que ce soit une arête ou une face de la frontière.

Laplacien de qualité

Aimé pour sa simplicité et sa rapidité, le lissage laplacien a été décliné en quelques versions légèrement plus lourdes, mais qui garantissent un plus grand gain en qualité. On se penchera exclusivement ici sur le lissage laplacien dit de qualité (*Quality Laplace*).

La différence entre le laplacien classique et de qualité est la manière de choisir la position finale du nœud. On remplacera l'utilisation d'un coefficient de frottement par une recherche linéaire le long de la droite qui passe par le nœud et le centre du voisinage élémentaire. On a choisi de tester huit positions sur cette droite. Pour chaque position, on évaluera la qualité que le voisinage élémentaire aurait si le nœud était déplacé à cet endroit. Puis, on sélectionnera la position qui offre la meilleure qualité, que ce soit la position d'origine du nœud ou l'une des positions sur la droite de recherche. Cette modification empêche tout déplacement du nœud dans le cas ou la droite de recherche ne permettrait pas d'amélioration. De cette manière, il est garanti que la qualité du maillage ne se détériore jamais au cours du lissage.

Versions riemanniennes du lissage laplacien

Bien que la version de qualité du lissage laplacien prenne en compte la qualité des éléments, la droite de recherche ne tient pas compte de la fonction métrique. Ce problème peut être réglé en modifiant la manière de calculer le « centre » des voisinages élémentaires.

Une première option consisterait à toujours considérer les arêtes émanant du nœud x comme des ressorts. Mais, cette fois-ci, ils exhiberaient un effet de répulsion si l'arête est plus courte que la taille spécifiée par la fonction métrique. Pour généraliser le nouvel algorithme aux espaces riemanniens, il suffirait de mesurer la taille des arêtes avec la bonne métrique. Bossen and Heckbert (1996) donnent quelques exemples de fonctions pour modéliser le comportement des ressorts. Toutefois, remarquez que les forces de répulsion rendent la méthode très instable lorsque le maillage est trop fin par rapport à la métrique spécifiée.

Alors, pour permettre l'adaptation de maillage par l'intermédiaire du lissage laplacien et pour éviter la très grande instabilité introduite par l'ajout de forces de répulsion, une nouvelle version du lissage laplacien a été développée dans le cadre de cette recherche. Il s'agit d'une généralisation directe aux espaces riemanniens de l'algorithme original. L'idée est de réécrire le calcul du centre du voisinage élémentaire c_P comme un *déplacement* moyen plutôt qu'une *position* moyenne.

Dans l'espace euclidien, on veut déplacer le nœud x vers le centre c_P de son voisinage élémentaire de sommets P_i :

$$c_P = \sum_{i=1}^{n} \frac{1}{n} P_i \tag{3.2}$$

Que l'on peut réécrire sous la forme d'un déplacement moyen :

$$c_P = x + \sum_{i=1}^{n} \frac{1}{n} (P_i - x) \tag{3.3}$$

Cette nouvelle représentation peut être généralisée en une somme pondérée :

$$c_P = x + \sum_{i=1}^{n} \bar{\lambda_i} (P_i - x), \text{ avec } \sum_{i=1}^{n} \bar{\lambda_i} = 1 \tag{3.4}$$

On utilisera la métrique spécifiée pour pondérer la contribution des nœuds voisins dans le déplacement moyen. Le poids λ_i est donné par le ratio entre la longueur du vecteur $P_i - x$ dans la métrique et sa norme dans l'espace euclidien. La longueur du vecteur est mesurée en échantillonnant une seule fois la métrique à mi-chemin entre le nœud x et son voisin P_i :

$$\lambda_i = \frac{l_M(P - x)}{\|P - x\|},\ \bar{\lambda_i} = \frac{\lambda_i}{\sum_{i=1}^{n} \lambda_i} \tag{3.5}$$

On remarque que si l'espace riemannien choisi correspond à l'espace euclidien, on retrouve la forme classique du lissage laplacien avec $\bar{\lambda_i} = 1/n$, puisque $\lambda_i = 1$, $\forall i \in [1..n]$. Effectivement, si l'espace riemannien utilisé correspond à l'espace euclidien, la longueur du vecteur $P_i - x$ dans la métrique sera toujours égale à sa norme dans l'espace euclidien.

Les implémentations des algorithmes laplacien à ressorts et laplacien de qualité utilisent cette somme pondérée pour construire leur droite de recherche. Bien que cette modification n'empêche pas la version à ressorts de générer des éléments inversés, elle permet de déplacer les nœuds de manière bien plus cohérente avec la fonction métrique.

Descente du gradient

Déplacer les nœuds en traitant le maillage comme un système de ressorts n'est qu'une heuristique ; l'efficacité de cette approche n'est soutenue par aucune démonstration géométrique. En fait, il est même facile de construire des cas où déplacer un nœud vers le centre de son voisinage élémentaire dégraderait la qualité de ce voisinage élémentaire (voir figure 2.7). On n'a qu'à penser aux nœuds près des frontières concaves dont le centre peut facilement se trouver à l'extérieur du domaine. En ajoutant une recherche linéaire au lissage laplacien, on a ouvert la porte à des approches beaucoup plus efficaces dont le comportement n'est plus seulement basé sur des heuristiques, mais également sur la qualité actuelle des voisinages élémentaires. Le terrain est maintenant prêt pour les algorithmes d'optimisation locale.

La descente du gradient est une technique d'optimisation locale très bien adaptée à notre problème de déplacement de nœuds. Notre fonction coût est naturellement donnée par la qualité du voisinage élémentaire, pour laquelle l'espace des paramètres est la position des nœuds dans l'espace euclidien. La première étape consiste à calculer le gradient de la fonction coût. Heureusement, nous avons pris soin de construire une fonction coût lisse en utilisant une moyenne pour définir la qualité des voisinages élémentaires. Si nous avions défini cette fonction à l'aide du minimum, elle aurait possédé une dérivée première discontinue, ce qui aurait entravé la convergence de l'algorithme. Donc, une fois le gradient calculé, on procède à une recherche linéaire dans sa direction. On distribue les positions à tester en fonction du rayon du voisinage élémentaire. Ce rayon est donné par la moyenne arithmétique des longueurs des arêtes qui émanent du nœud traité, tel qu'illustré à la figure 3.6. Le rayon du voisinage élémentaire permet de calibrer les déplacements pour éviter de toujours tester des positions hors du voisinage élémentaire (déplacements trop grands) ou d'obtenir une

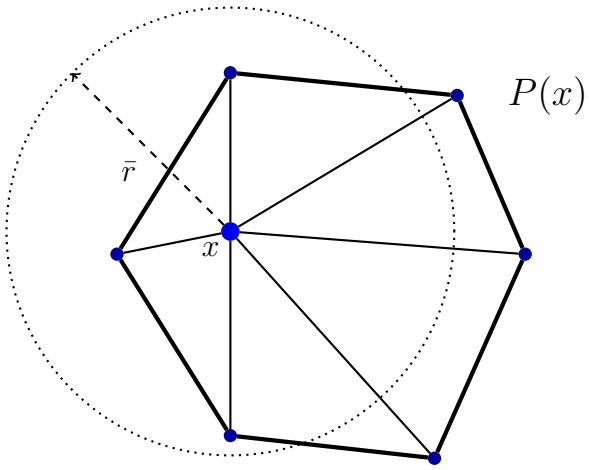


Figure 3.6 Le rayon $\bar{r}$ du voisinage élémentaire de x

convergence trop lente (déplacements trop petits). On teste encore une fois 8 positions le long de la droite de recherche. Il reste finalement à déterminer le meilleur candidat entre les positions testées et à déplacer le nœud à cet endroit. On observera au chapitre 5 que la droite de recherche proposée par le gradient permet un gain en qualité bien supérieure à la droite définie par le centre du voisinage élémentaire.

Nelder-Mead

Si la fonction coût n'avait pas été lisse, par exemple par l'utilisation du minimum des qualités comme il est commun de le faire, il aurait été préférable de choisir une technique d'optimisation locale indépendante du gradient. L'algorithme de Nelder-Mead fait partie de cette catégorie. De plus, celui-ci a déjà été utilisé avec succès dans l'implémentation d'algorithmes de déplacement de nœuds sur GPU, montrant ainsi sa compatibilité avec le modèle d'exécution SIMD des processeurs graphiques.

Malgré la présence importante de branchements conditionnels, cet algorithme démontre d'excellentes performances sur GPU pour des gains en qualité comparables à la descente du gradient. C'est pourquoi, bien que notre fonction coût soit lisse, il a été jugé pertinent d'implémenter cet algorithme également.

On aura recours une seconde fois au rayon du voisinage élémentaire pour initialiser cet algorithme. Ce rayon permettra ici de dimensionner le simplexe de recherche proportionnellement à la taille des éléments. Le reste de l'algorithme est présenté à la section 2.4.2. Il faut évidemment projeter les nœuds frontières sur leur support topologique une fois la position optimale déterminée. Les différents paramètres de l'algorithme, tels que les ratios de réflexion, contrac-

tion, dilatation, et rétrécissement correspondent aux valeurs standards (Singer and Nelder (2009)). Les conditions de terminaison par convergence spatiale et de valeur ont été choisies par une série de tests numériques. Les valeurs sélectionnées sont les suivantes :

$$\begin{cases} \alpha = 1.0 & (R\acute{e}flexion) \\ \beta = 0.5 & (Contraction) \\ \gamma = 2.0 & (Dilatation) \\ \delta = 0.5 & (R\acute{e}tr\acute{e}cissement) \end{cases} \quad \begin{cases} \text{Convergence de valeur} = 1.0e-4 \\ \text{Convergence spatiale} = 0.0 \\ \text{Nombre de déplacements} = 12 \end{cases} \tag{3.6}$$

La convergence de valeur est donnée par la différence entre la qualité du meilleur et du pire sommet du simplex de recherche. Actuellement, aucun test de convergence spatiale n'a été intégré à l'algorithme, d'où la valeur 0.0 pour indiquer une précision infinie. On repose donc sur le nombre maximal de déplacements pour terminer l'algorithme en cas de divergence. Remarquez qu'implémenter la détection de la convergence spatiale n'est pas nécessairement souhaitable. Cette détection requiert le calcul du volume du simplex de recherche, ce qui représente une quantité non négligeable de calculs supplémentaires à chaque itération de l'algorithme.

On notera également que le classement des sommets du simplex de recherche en ordre de qualité est maintenu manuellement. Puisqu'il n'y a que quatre sommets à classer, l'algorithme débute par un petit tri par bulles. À chaque itération, le classement est mis à jour en effectuant un cycle de tri par insertion. Le nouveau sommet est inséré et le pire des cinq sommets est retiré de la liste.

3.1.6 Les topologistes

Les topologistes sont responsables des modifications topologiques. De la même manière que les lisseurs, ils utilisent les modules d'échantillonnage, de mesure et d'évaluation pour pouvoir prendre des décisions. Il existe plusieurs types d'opérations topologiques, mais puisqu'ils sont complémentaires, ils doivent tous être utilisés de concert. C'est pourquoi un seul topologiste a été développé pour ce module. Ce topologiste est largement inspiré de la procédure BATR de Freitag and Ollivier-Gooch (1997) et a été implémenté en partie par programmation dynamique tel que proposé par Shewchuk (2002b). Seules les opérations topologiques manipulant des tétraèdres ont été implémentées. Il existe dans la littérature des opérations pour les maillages hexaédriques, mais leur application est plus complexe et la totalité des opérations atomiques n'a pas encore été découverte (voir Wang et al. (2016)).

Tout d'abord, avant d'appliquer les principales opérations topologiques, on tentera de répa-

rer certains défauts typiquement retrouvés sur les frontières du domaine. On appliquera trois types de réparation successivement. Premièrement, on supprimera les chapeaux. On nomme chapeaux les tétraèdres reposant sur la frontière dont l'un des sommets n'est relié à aucun autre tétraèdre. Ces éléments peuvent être supprimés tout simplement sans que cela affecte la conformité du maillage. Deuxièmement, on procédera à la suppression des tétraèdres dont l'une des arêtes n'est partagée par aucun autre tétraèdre. Encore une fois, supprimer ces tétraèdres n'affecte pas la conformité du maillage. Finalement, pour les tétraèdres qui comptent trois sommets sur une même frontière, on vérifiera si en projetant le quatrième sommet sur cette frontière on ne pourrait pas améliorer la qualité des éléments environnants. Si c'est le cas, on détruira le tétraèdre en question et l'on contraindra le quatrième sommet à résider sur la frontière. Ces trois types de réparation rendent le reste du processus d'adaptation par modifications topologiques beaucoup plus robuste et permettent un bien meilleur gain en qualité près des frontières du domaine.

Le topologiste applique les différentes modifications topologiques dans un ordre fixe à l'aide de deux niveaux de boucles imbriquées. Le premier niveau permet d'alterner entre les différents types de modifications topologiques. Le deuxième niveau permet d'appliquer une même opération à plusieurs reprises sur tous les éléments du maillage. L'ordre dans lequel les types d'opérations sont appliqués est arbitraire. Dans notre cas, avant même d'entrer dans la boucle principale, on exécute la vérification et la correction des défauts de frontière. On applique ensuite le retournement de face jusqu'à ce qu'il ne soit plus possible d'augmenter la qualité du maillage à l'aide de cette opération. Lorsqu'un retournement de face est effectué, trois nouvelles faces sont créées autour de la nouvelle arête. Ces faces doivent être visitées à leur tour avant de passer au retournement d'arêtes. Puis on applique le retournement d'arêtes en s'assurant de gérer le même type de réaction en chaîne que dans le retournement de faces. Après avoir complété les deux types de retournement, on termine la boucle principale par le raffinement d'arêtes et la fusion de nœuds.

Les nombres d'itérations globales, de raffinements d'arêtes et de fusion de nœuds sont choisis par heuristique. Des valeurs par défauts sont fournies, mais elles peuvent facilement être modifiées par l'usager au lancement des algorithmes.

3.2 Algorithme global d'adaptation

Déterminer à quel moment le processus d'adaptation a convergé est une tâche complexe. D'une part, notre discussion de la section 2.3 nous rappelle qu'il n'existe pas de mesure absolue ni même de consensus à l'égard d'une mesure standard pour évaluer la qualité d'un maillage. Sans compter qu'on exprime souvent cette qualité à l'aide de deux valeurs : l'une

pour le pire élément et l'autre pour la moyenne des qualités. D'autre part, nous devons définir un seuil de convergence. Ce seuil devrait-il être absolu, configurable ou même dépendant de certaines propriétés du maillage ? Ces incertitudes compliquent l'implémentation d'un algorithme d'adaptation automatisé.

Il faut également considérer toutes les boucles dont le nombre d'itérations est variable :

- Boucle globale d'adaptation
  - Boucle globale de modifications topologiques
    - Retournement de faces (dynamique)
    - Retournement d'arêtes (dynamique)
    - Raffinement d'arêtes/Fusion de nœuds
  - Boucle globale de déplacement de nœuds

Dans cet arbre de boucles imbriquées, seules deux boucles terminent de manière automatique : les boucles de retournement de faces et d'arêtes. C'est-à-dire qu'elles terminent en un nombre fini d'itérations sans l'aide d'un seuil maximal. La boucle de raffinement d'arêtes/fusion de nœuds termine parfois lorsque le maillage ne peut plus être modifié, mais ce n'est pas toujours le cas. Il existe des configurations où, en deux ou trois opérations de raffinement/fusion, le maillage retrouve sa configuration originale et le processus entre dans une boucle infinie. Pour éviter ce problème, on préférera reposer sur un nombre maximal d'itérations.

À chaque itération de la boucle globale de modifications topologiques, on dénombre la quantité d'opérations effectuées pour chaque type d'opération. Si le nombre total d'opérations effectuées est nul, le processus de modification topologique a convergé. De plus, si le nombre d'opérations est petit (< 5) et égal au nombre d'opérations de l'itération précédente, il est raisonnable de croire que le processus est entré dans une boucle infinie. Dans ce cas, on forcera la terminaison du processus de modification topologique. Finalement, on forcera la terminaison du processus si le nombre maximal d'itérations globales est atteint. Ces seuils peuvent être configurés par l'usager dans l'interface graphique.

Contrairement aux algorithmes de modifications topologiques qui opèrent dans un espace discret, la convergence des algorithmes de déplacement de nœuds ne peut être basée sur le nombre d'opérations complétées. On se basera plutôt sur le gain en qualité obtenu à chaque itération. Des deux mesures de la qualité du maillage, soit le minimum et la moyenne harmonique des qualités, seule la moyenne croît de manière monotone. Cela dépend bien sûr de la manière dont la qualité des voisinages élémentaires est mesurée, mais il arrive que la qualité du pire élément se détériore momentanément pour s'améliorer bien davantage par la suite. Le seuil de convergence sera donc appliqué à la moyenne des qualités. La valeur du seuil est arbitraire et peut actuellement être définie par l'usager dans l'interface graphique.

Bien qu'il soit profitable pour un utilisateur moyen de reposer sur un algorithme d'adaptation automatisé, il est préférable dans le cadre de cette recherche d'avoir le maximum de contrôle sur le nombre d'itérations exécutées. On peut ainsi déterminer la capacité des algorithmes à obtenir un certain gain de qualité en un certain nombre d'itérations donné. On pourra également comparer les vitesses d'exécution de ces algorithmes sur la base d'un nombre fixe d'itérations. C'est pourquoi les nombres d'itérations de la boucle globale d'adaptation et de déplacement de nœuds peuvent également être fixés.

La liste suivante donne les compteurs à éditer dans l'interface graphique (figure 3.7) pour configurer le nombre d'itérations voulues pour chacune des boucles d'adaptation :

— Boucle globale d'adaptation : *Scheduling -> Global pass count*
— Boucle globale de modifications topologiques : *Topological Modifications -> Pass count*
— Retournement de faces : *[toujours calculé dynamiquement]*
— Retournement d'arêtes : *[toujours calculé dynamiquement]*
— Raffinement d'arêtes/Fusion de nœuds : *Topological Modifications -> Refinement sweeps*
— Boucle globale de déplacement de nœuds : *Node Relocation -> Pass count*

3.3 Structure de données CPU

Les structures de données utilisées pour représenter le maillage en C++ ne sont pas particulièrement complexes. Elles ont d'abord été développées pour être les plus uniformes possible avec leurs pendants GPU, ce qui nous limite à l'utilisation de structures et de tableaux C. On se retrouve donc avec six tableaux de structures : les nœuds, les dictionnaires topologiques, les tétraèdres, les pyramides, les prismes et les hexaèdres. Ces tableaux se référencent mutuellement à l'aide d'indices. Par exemple, chaque hexaèdre contient la liste des indices de ses huit sommets dont on peut retrouver la position dans le tableau de nœuds.

Le maillage en C++ contient également la représentation de la frontière (aussi appelé modèle géométrique). Le modèle géométrique est utilisé pour contraindre le mouvement des nœuds qui reposent sur la frontière du domaine. Lors du raffinement d'arêtes et de la fusion de nœuds, il est possible de déterminer l'appartenance du nœud résultant à l'un des supports topologiques de la frontière selon l'appartenance des deux nœuds à l'origine de l'opération. Cette fonctionnalité est nécessaire pour garantir la conformité du maillage lors des modifications topologiques. Puisqu'aucune modification topologique ni aucun déplacement de nœud frontière ne sont effectués sur le GPU, la représentation de la frontière n'existe qu'en C++ sur le CPU.

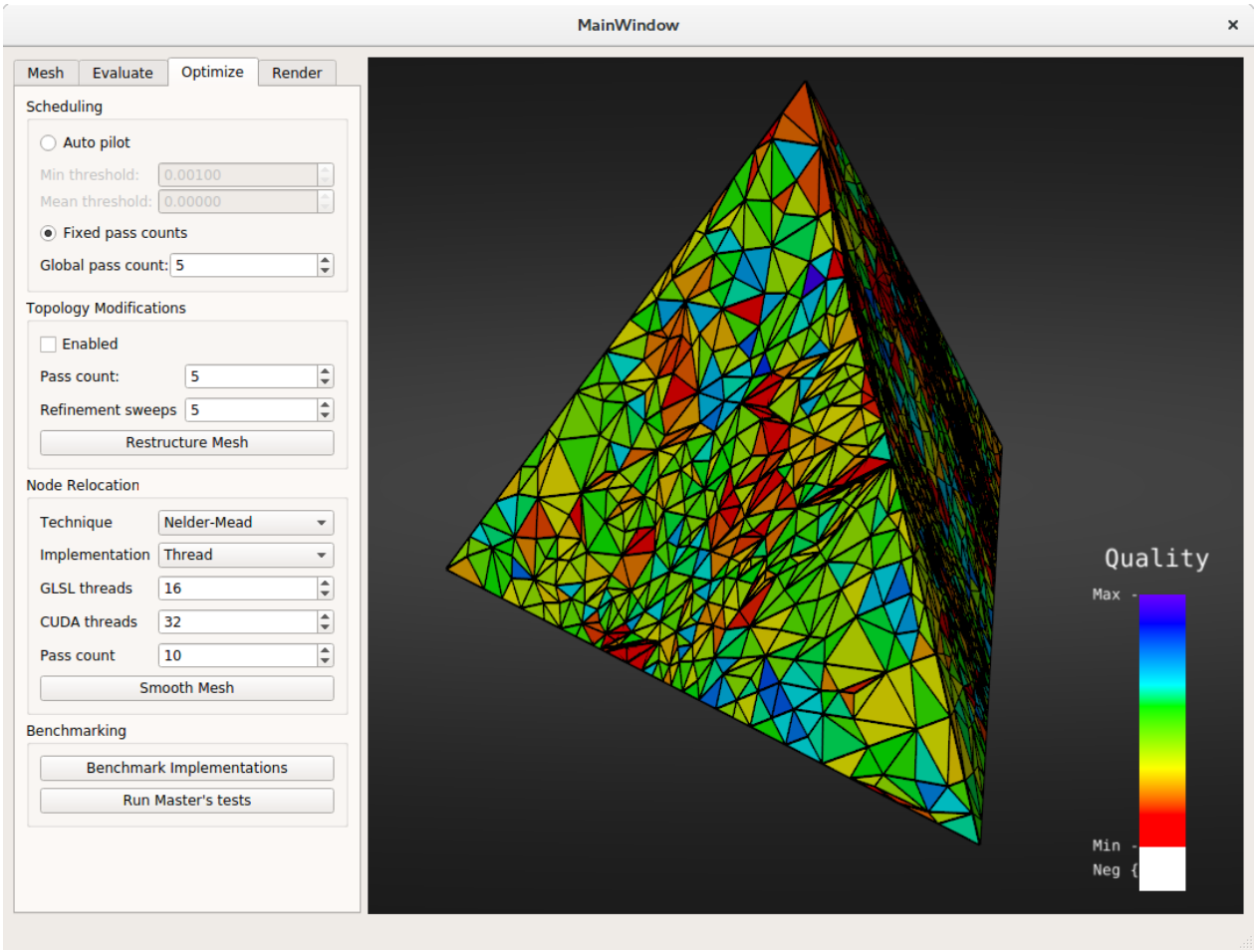


Figure 3.7 L'interface graphique permet de configurer tout le processus d'adaptation

3.3.1 Structures de données minimales

Notre but étant d'adapter un maillage volumique à l'aide d'un champ de tenseurs métriques, notre maillage doit pouvoir contenir : la position et le support topologique de chaque nœud et les sommets des tétraèdres, pyramides, prismes et hexaèdres. À partir de ces informations, on est en mesure de construire les dictionnaires topologiques qui accélèrent la recherche des nœuds voisins et des éléments voisins. Ces dictionnaires topologiques seront largement utilisés par les algorithmes de déplacement de nœuds et de modifications topologiques.

Prenez note que les supports topologiques et les dictionnaires topologiques sont deux types d'entités complètement différents. Les supports topologiques sont les sommets, les arêtes, les faces et les volumes qui composent la représentation du domaine et de sa frontière. Les dictionnaires topologiques sont de petites structures de données qui nous renseignent sur la

connectivité entre un nœud et son entourage. Un dictionnaire topologique nous renseigne également sur le support topologique auquel appartient un nœud donné.

On ajoutera un dernier type de donnée au maillage. Puisque l'interpolation de la métrique sur un maillage de fond tétraédrique est l'une de nos principales méthodes d'échantillonnage, on garde des indices supplémentaires pour chacun des nœuds et des éléments afin d'estimer leur position dans le maillage de fond. Ces indices mémorisent le dernier tétraèdre du maillage de fond sur lequel la métrique a été interpolée pendant le processus d'adaptation. L'ensemble des structures de données manipulées par le logiciel d'adaptation est présenté à la figure 3.8 sous forme de diagramme de classes.

Éventuellement, lorsque le processus d'adaptation aura convergé, on voudra interpoler la solution précédente sur le maillage adapté. Cette opération dépasse toutefois le cadre de la présente recherche. De plus, plusieurs logiciels proposent déjà cette fonctionnalité. Pour pouvoir en profiter, il suffira d'exporter le maillage final dans le bon format. Pour cette raison, les variables de la solution (vitesse, pression, température, etc.) sont totalement ignorées par le logiciel d'adaptation. Puisque la fonction métrique est construite à partir de la solution, cela signifie également que la métrique doit être générée préalablement au processus d'adaptation.

3.3.2 Représentation des frontières

La représentation des frontières est fournie par un logiciel de CAO. Le logiciel d'adaptation ne supporte actuellement que les modèles composés d'un unique volume borné par des sommets, des droites, des cercles, des plans, des cylindres et des sphères. Les supports topologiques sont divisés en quatre catégories selon leur dimension : les sommets (0D), les arêtes (1D), les faces (2D) et les volumes (3D). Depuis un support topologique, il est possible de visiter les autres supports de dimension différente qui lui sont directement connectés. Par exemple, depuis un sommet, il est possible de visiter les arêtes et les faces que celui-ci borne. Cette fonctionnalité est nécessaire pour déterminer le support topologique du résultat d'une fusion de nœuds et d'un raffinement d'arête.

L'ensemble des supports topologiques est assemblé à l'intérieur d'une structure représentant le domaine du problème. Actuellement, tous les domaines sont assemblés manuellement en C++. Il serait possible toutefois d'intégrer un module d'importation pour les fichiers IGES ou STEP.

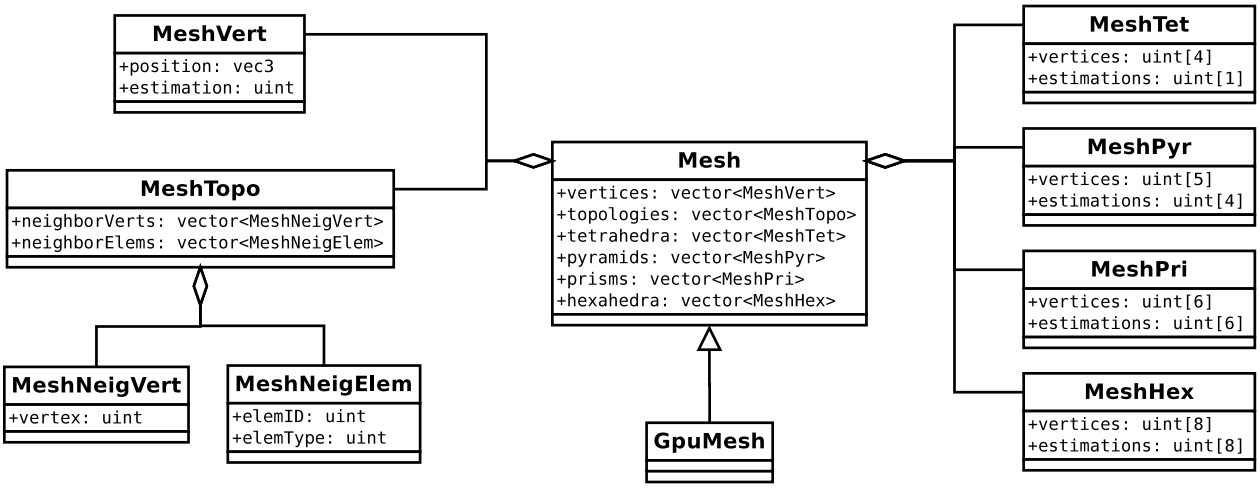


Figure 3.8 Ensemble des structures de données utilisées pour représenter un maillage

3.3.3 Tableau de nœuds (*MeshVert*)

La représentation du maillage commence réellement avec le tableau de nœuds. Ce tableau contient la position des nœuds ainsi que l'estimation du tétraèdre où se trouve le nœud dans le maillage de fond. Cette estimation n'est pas toujours exacte puisque les nœuds se déplacent progressivement au cours du processus d'adaptation. Cependant, elle accélère grandement la recherche de sa vraie position. Cette estimation est utilisée lors du calcul la longueur des arêtes et mise à jour à chacune de ses utilisations.

3.3.4 Tableau de dictionnaires topologiques (*MeshTopo*)

À chaque nœud est associé un dictionnaire topologique qui liste tous les nœuds voisins avec lequel celui-ci partage une arête et tous les éléments voisins qui l'utilisent comme sommet. La liste des nœuds voisins sert à mesurer le « rayon » du voisinage élémentaire. La liste des éléments voisins est utilisée pour déterminer le centre et la qualité du voisinage élémentaire. Les dictionnaires topologiques spécifient également le support topologique du nœud. Les supports topologiques possèdent une fonction pour projeter le nœud sur la partie de la frontière qu'ils représentent. Les opérations topologiques utilisent aussi les supports topologiques pour déterminer si une opération est permise entre deux nœuds donnés. Par exemple, la fusion de deux nœuds reposant sur des supports topologiques non connectés est interdite, car elle affecterait la conformité du maillage.

3.3.5 Tableaux d'éléments (*MeshTet*, *MeshPyr*, *MeshPri* et *MeshHex*)

Le maillage contient un tableau pour chaque type d'éléments. Chaque élément est représenté par une petite structure qui liste les indices de ses sommets dans le tableau de nœuds et les estimations des positions des coins dans le maillage de fond pour l'échantillonnage par recherche locale. Par coin, on entend les mêmes coins qui ont été définis pour les mesures de qualité (voir 2.3.3). Ces estimations seront justement utilisées pour échantillonner la fonction métrique pendant l'évaluation de la qualité. La raison pour laquelle on utilise une estimation par coin et non une estimation pour tout l'élément est que l'on veut diminuer les déplacements inutiles dans le maillage de fond au prix d'un peu d'espace mémoire.

3.3.6 Ensembles de nœuds indépendants

Une sous-structure de données du maillage stocke les ensembles de nœuds indépendants, c'est-à-dire les groupes de nœuds dont on permet le déplacement simultané. Si le déplacement des nœuds est réalisé entièrement sur le CPU, cette structure se résume à un tableau en deux dimensions. La première dimension représente les différents ensembles de nœuds indépendants. La deuxième dimension représente les fils d'exécution qui déplaceront les nœuds. Chacune des cellules de ce tableau contient la liste des nœuds en fonction de l'ensemble de nœuds indépendants auquel ils appartiennent et du fil d'exécution auquel ils sont assignés. La figure 3.9 illustre une distribution fictive pour huit groupes de nœuds indépendants et quatre fils d'exécution. Le nombre de nœuds contenus dans chaque liste est écrit dans les cellules.

On verra à la section 4.4 que la structure gagnera énormément en complexité lorsqu'on y intégrera la distribution des nœuds entre les différents cœurs CPU et GPU.

		Ensembles de nœuds							
		1	2	3	4	5	6	7	8
Threads	1	242	345	345	242	338	338	225	225
	2	242	345	345	241	338	338	225	225
	3	241	345	345	241	337	337	225	225
	4	241	345	345	241	337	337	225	225

Figure 3.9 Distribution de nœuds pour huit ensembles et quatre fils d'exécution

CHAPITRE 4 INFRASTRUCTURE GPU

L'infrastructure du logiciel d'adaptation présenté jusqu'ici possède une couche supplémentaire pour porter les structures de données et les algorithmes au GPU. Cette couche offre des extensions aux classes existantes. Elle permet d'une part de traduire et d'envoyer les structures de données sur le GPU et d'autre part de synchroniser l'exécution des noyaux de calcul avec les fils d'exécution du CPU.

Dans le présent chapitre, on présentera une nouvelle méthode pour échantillonner la métrique qui est beaucoup mieux adaptée à la réalité des GPU. On abordera ensuite la stratégie adoptée pour paralléliser le déplacement de nœuds sur le GPU. Cette stratégie implique le partitionnement des nœuds selon une nouvelle classification ainsi qu'une répartition des tâches entre processeurs. On présentera également quatre nouvelles versions des algorithmes de déplacement de nœuds développées spécifiquement pour les processeurs SIMD. On terminera par la traduction des structures de données et la synchronisation des processeurs.

4.1 Échantillonnage de la métrique sur GPU

Échantillonner la métrique sous forme de fonction analytique se fait aussi bien sur le CPU que le GPU. Malheureusement, on ne peut qu'obtenir des résultats théoriques en représentant la métrique spécifiée de cette manière. La méthode habituelle de représenter une métrique en ingénierie consiste à stocker les tenseurs métriques aux nœuds du maillage de fond et de procéder par recherche locale pour échantillonner la métrique en un point. Bien que cette méthode soit très efficace sur le CPU, elle semble presque impraticable sur le GPU. C'est pourquoi on étudiera une nouvelle représentation de la métrique : la texture.

4.1.1 Divergence d'exécution induite par le maillage de fond

En mesurant la précision des estimateurs utilisés par la *recherche locale*, on remarque que la majorité de ceux-ci identifient directement le bon élément du maillage de fond. Une minorité est décalée de plus d'une dizaine d'éléments. Les estimateurs qui ne permettent pas de rejoindre l'échantillon demandé sont des cas d'exception. En moyenne, on observe donc que les estimateurs engendrent un peu moins d'un déplacement par échantillon, ce qui correspond à la complexité algorithmique théorique de $\mathcal{O}(1)$ avancée à la section 3.1.2.

Bien que ces statistiques soient toujours valides lorsque la recherche locale est effectuée sur le GPU, on constate que l'impact des estimateurs problématiques est multiplié. Cette am-

plification est due au modèle SIMD des GPU. Lorsqu'un échantillon est rejoint en plusieurs déplacements depuis un estimateur, ce sont tous les fils d'exécution du *warp* qui sont mis sur pause en attendant que l'échantillon soit rejoint. Il suffit qu'un estimateur sur 32 soit problématique pour que les *warps* doivent en moyenne patienter une dizaine de déplacements avant de poursuivre leur exécution.

On se rappellera que l'échantillonnage de la métrique est l'opération la plus déterminante sur le temps de calcul du processus d'adaptation. Calculons par exemple le nombre d'échantillons lus lors de l'évaluation de la qualité d'un voisinage élémentaire de 8 hexaèdres. Chaque hexaèdre possède 8 coins pour lesquels on estimera la métrique par une quadrature de Gauss en 4 points. Pour évaluer la qualité de ce voisinage élémentaire, on échantillonnera donc la métrique à l'aide de (8 hexaèdres $\times$ 8 coins $\times$ 4 échantillons) = 256 échantillons. On se rappellera également qu'une itération de la *descente du gradient* calcule la qualité d'un voisinage élémentaire en 14 endroits, ce qui représente un total de 3584 échantillons pour déplacer un seul nœud.

Si autant d'échantillons sont requis pour déplacer chaque nœud du maillage, il est impératif de trouver une technique plus efficace si l'on veut rendre les algorithmes de déplacement de nœuds compétitifs sur le GPU. Dans le domaine du rendu en temps réel (ex. : jeux vidéos et visualisation interactive), une attention particulière a été portée à l'échantillonnage de texture pour offrir des débits de lecture en mémoire toujours plus élevés. Puisqu'échantillonner une texture ne produit jamais de divergence dans un noyau de calcul, représenter la métrique spécifiée par une texture 3D semble être une solution prometteuse.

4.1.2 Discrétisation dans une grille uniforme (texture)

Jusqu'à présent, nous n'avions qu'une seule méthode pour échantillonner la fonction métrique dans la pratique : la recherche locale. L'échantillonnage par recherche locale et par texture possèdent tous deux un coût en temps de calcul de l'ordre de $\mathcal{O}(1)$, mais l'échantillonnage d'une texture à l'avantage d'être beaucoup mieux adapté au modèle d'exécution SIMD et est toujours exécuté matériellement par les cartes graphiques.

Par contre, cette méthode implique un compromis important entre la taille et la précision de la représentation. Une texture est essentiellement une grille uniforme. Pour les problèmes tridimensionnels, les tenseurs métriques sont représentés par des matrices 3x3, toujours symétriques. Cette symétrie nous permet de ne stocker que les 6 valeurs non redondantes, ce qui représente une économie en espace mémoire de 33%. Puisque les textures sont échantillonnées et interpolées matériellement par les cartes graphiques, elles sont limitées à 4 composantes par cellule (texel). Pour représenter un tenseur métrique, on utilise donc deux textures de 3

composantes chacune. Les cartes graphiques les plus récentes supportent des textures en virgule flottante, ce qui permet de stocker directement les composantes des tenseurs métriques sans avoir à passer par une représentation en virgule fixe. La décomposition du tenseur métrique est illustrée à la figure 4.1 sous forme matricielle. Dans les noyaux de calcul, le tenseur métrique est réassemblé en échantillonnant les deux textures puis en reconstruisant la matrice en plaçant les composantes aux bons endroits.

Il y a deux aspects auxquels il faut faire attention en initialisant les textures. D'abord, il y aura nécessairement des cellules qui couvriront plus d'un nœud. Dans ce cas, quelle valeur devrions-nous stocker ? Plusieurs choix s'offrent à nous tels : interpoler la métrique sur le maillage de fond au centre de la cellule, choisir la métrique la plus restrictive sur le domaine couvert par la cellule, utiliser une quadrature de gauss pour obtenir une approximation polynomiale, etc. L'autre aspect important est d'assurer la validité des échantillons près des frontières. À la création des textures, les cellules ont des valeurs aléatoires. On peut initialiser les textures en assignant une valeur arbitraire à toutes les cellules (la matrice identité par exemple), mais cette valeur a peu de chance d'avoir du sens localement, peu importe le cas. Le problème devient plus important lorsqu'on active l'interpolation linéaire automatique des textures. Dans ce cas, près des frontières, les échantillons seront toujours interpolés avec des cellules qui ne couvrent aucune partie du domaine et pour lesquelles la métrique spécifiée est indéfinie. Pour éviter les comportements erratiques, on s'assurera d'initialiser une couche supplémentaire de cellules autour du domaine en échantillonnant la métrique par projection sur la frontière du domaine. Une tranche de texture initialisée par la valeur au centre avec une couche supplémentaire est donnée à la figure 4.2. C'est cette méthode qui est implémentée dans l'adaptateur.

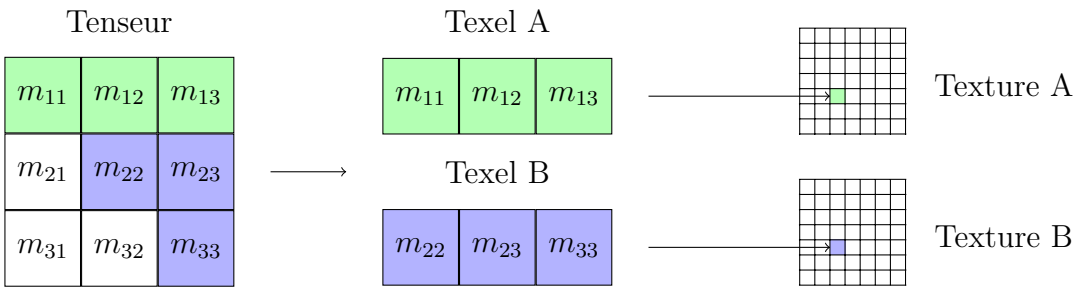
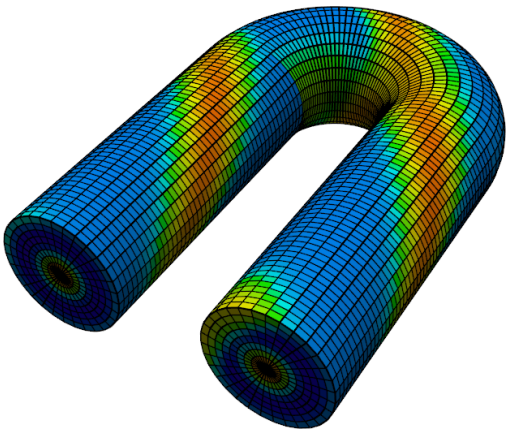
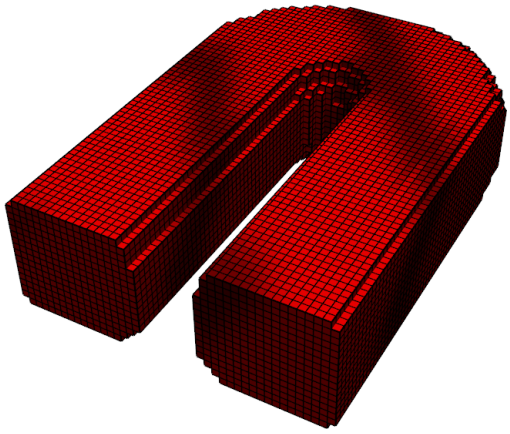


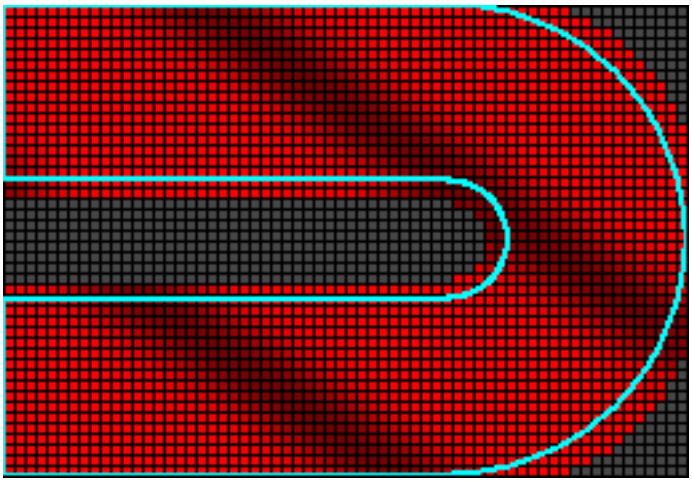
Figure 4.1 Stockage de la métrique sous forme de deux textures 3D



(a) Maillage



(b) Texture 3D



(c) Tranche de la texture à z=0.0 (géométrie en turquoise)

Figure 4.2 Maillage et composantes m_{11} des tenseurs métriques stockés sous forme de texture

4.1.3 Convergence de la nouvelle représentation

Le principal défaut des textures est leur nature uniforme. Le but de l'adaptation de maillage est de réduire le nombre d'éléments tout en respectant un certain niveau de précision dans la représentation de la solution. Autrement dit, adapter un maillage revient à manipuler géométriquement et topologiquement la grille de calcul pour que la taille et la forme de ses éléments épousent le comportement local de la solution. De ce point de vue, on peut dire qu'une texture est nécessairement « inadaptée » à la solution. Ce défaut nous pousse à faire un compromis entre la taille des textures utilisées et la précision spatiale qu'elles offrent.

Pour prédire la précision de la métrique sous forme de texture, on peut se baser sur les résultats empiriques de Labbé et al. (2011). D'après leurs résultats, autant pour les maillages uniformes, adaptés isotropes et adaptés anisotropes, les auteurs concluent que l'erreur d'interpolation diminue quadratiquement en fonction du nombre d'éléments. Toutefois, pour le problème de réaction-diffusion étudié dans cet article, les maillages adaptés anisotropes offrent la même précision que les maillages uniformes tout en utilisant près de 600 fois moins d'éléments, ce qui représente d'importantes économies en espace mémoire.

Toutefois, notre contexte est sensiblement différent. Notre but n'est pas de représenter fidèlement une solution, mais bien de fournir une métrique spécifiée adéquate pour adapter notre maillage au comportement de la solution. Les imprécisions dans la représentation de la métrique ont un impact beaucoup moins critique sur les résultats de la simulation. On prendra le temps au chapitre 5 d'analyser l'impact de la résolution des textures sur la qualité des maillages adaptés.

4.2 Lissage coopératif CPU-GPU

Parce que la représentation du domaine et le déplacement des nœuds frontières représentaient des défis trop importants sur le GPU, la décision d'abstraire complètement cette partie des noyaux de calcul a été prise. On dira des noyaux de calcul qu'ils sont agnostiques face aux frontières au sens où ils sont « conscients » de leur existence, mais ils opèrent sans s'en soucier. Nous ne leur ferons pas déplacer les nœuds frontières, mais constatons que le processus d'adaptation ne peut être complet sans que ceux-ci soient également déplacés. Puisque tous les algorithmes possèdent à la base une implémentation en C++, on distribuera les nœuds entre processeurs en assignant les nœuds frontières au CPU et tous les autres nœuds au GPU.

Puisque le travail est réparti sur le processeur principal (CPU) et un coprocesseur (GPU), certains nœuds devront être copiés d'une mémoire à l'autre chaque fois qu'un ensemble de nœuds indépendants sera traité. En plus d'être une opération coûteuse, cette copie est

exécutée à une très haute fréquence tout au long du processus de lissage. Afin de réduire l'impact de ces copies sur le temps de calcul, il est possible de réorganiser le tableau de nœuds. Ainsi, le volume de données copié et sa fragmentation en mémoire seront minimisés. L'astuce consiste à structurer le tableau de nœuds selon deux niveaux de classification. Cette classification et son implémentation seront présentées dans cette section.

4.2.1 Classification des nœuds

On sait déjà que l'on doit partitionner les nœuds du maillage en ensembles de nœuds indépendants. Pour distribuer le déplacement des nœuds entre CPU et GPU, on introduira un second partitionnement orthogonal au premier qui est basé sur les positionnements topologiques des nœuds. Quatre nouvelles catégories sont définies : les nœuds fixes, frontières, sous-surfaciques et intérieurs. Les nœuds fixes reposent sur les sommets de la frontière et ne sont jamais déplacés. Les nœuds frontières reposent sur les arêtes et les faces de la frontière et sont uniquement déplacés par le CPU. Les nœuds sous-surfaciques sont à l'intérieur du domaine, mais la qualité de leur voisinage élémentaire dépend d'au moins un nœud frontière. Ils peuvent être déplacés par le GPU sans problème, bien que celui-ci n'ait aucune connaissance de la géométrie de la frontière. Toutefois, les nœuds sous-surfaciques ont la particularité d'être essentiels aux nœuds frontières et aux nœuds intérieurs. C'est-à-dire qu'à chaque fois que l'on déplace un nœud sous-surfacique, sa nouvelle position doit être communiquée à tous ses voisins, qu'ils soient sur le CPU ou le GPU. Les nœuds dits « intérieurs » n'ont quant à eux pas ce besoin, car tous leurs voisins sont déplacés sur le GPU. Ils peuvent donc résider plusieurs itérations sur le GPU sans que leur nouvelle position soit communiquée au CPU. Un exemple de ce partitionnement est présenté à la figure 4.3.

Le regroupement des nœuds du maillage selon deux partitionnements orthogonaux produit une structure particulièrement complexe à l'intérieur même du tableau de nœuds (voir 4.4). Regrouper les nœuds selon leur catégorie à l'intérieur du tableau de nœuds n'est pas une nécessité, mais cela accélère énormément les transactions mémoires entre le CPU et le GPU. De cette manière, chaque fois qu'un groupe indépendant est traité, tous les nœuds déplacés sur le CPU dont la position doit être mise à jour sur le GPU (nœuds frontières) et tous les nœuds déplacés sur le GPU dont la position doit être mise à jour sur le CPU (nœuds sous-surfaciques) forment des segments contigus en mémoire. Donc, à l'aide de deux copies, tous les nœuds déplacés et uniquement ces nœuds sont copiés. Dans le contexte d'une mémoire non partagée entre CPU et GPU, il s'agit d'une utilisation optimale de la bande passante.

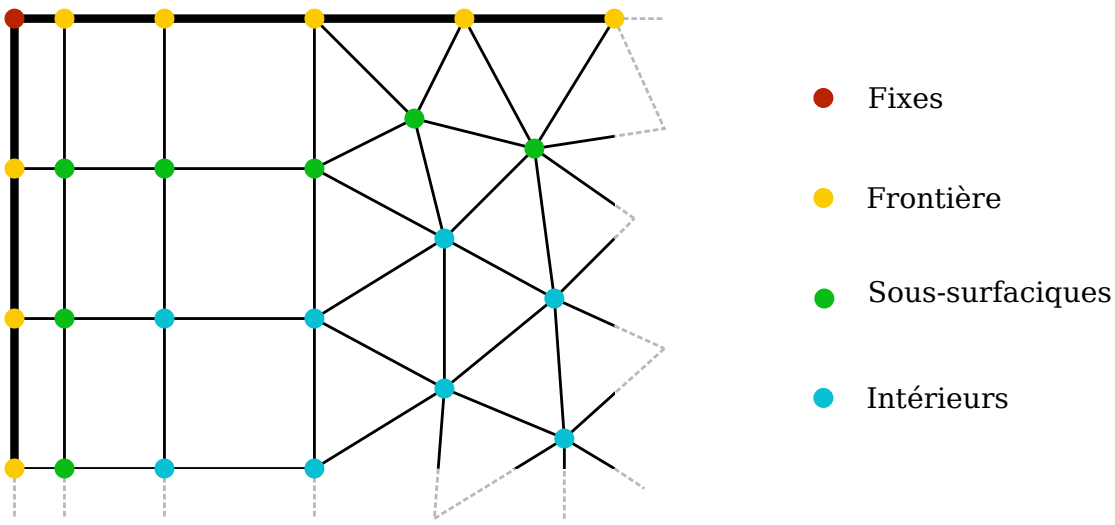


Figure 4.3 Maillage 2D composé de triangles et de quadrangles dont les nœuds ont été classés selon leur position topologique

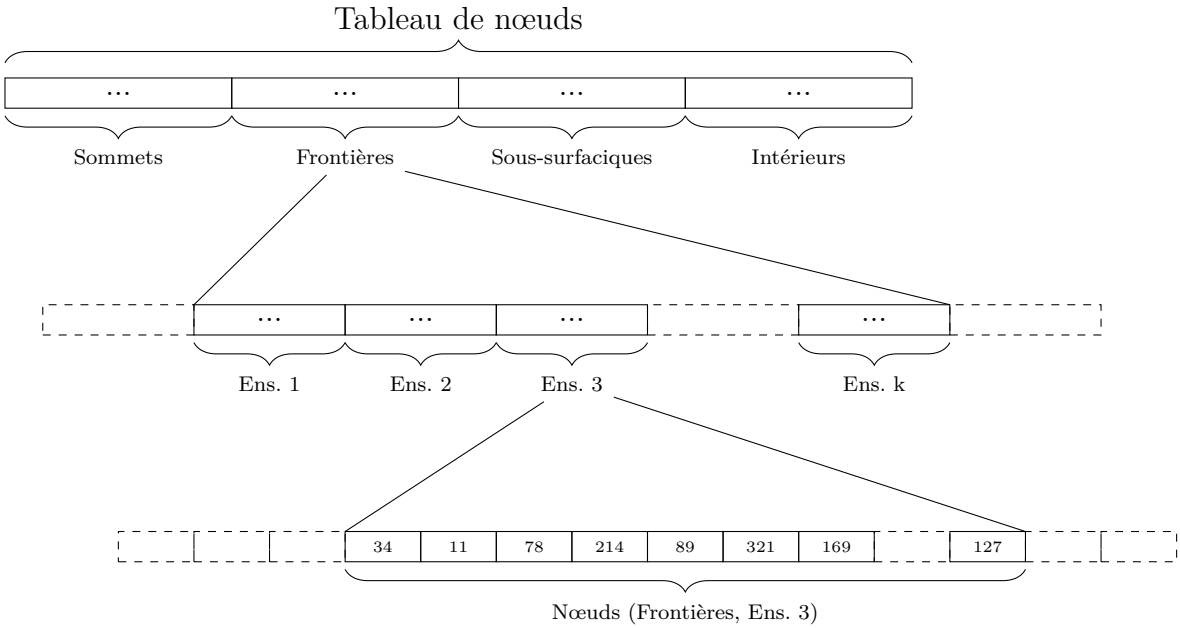


Figure 4.4 Le tableau de nœuds du maillage est d'abord divisé en positions topologiques puis sous-divisé en groupes indépendants

4.2.2 Validité de la stratégie

On peut se demander pourquoi il est possible de déplacer les nœuds sous-surfaciques sur le GPU si ce processeur n'a aucune connaissance de la frontière. Si une partie du voisinage élémentaire de ces nœuds réside sur la frontière, ne sera-t-il pas possible de créer des éléments invalides ou d'affecter la conformité du maillage ? La réponse est *non* puisque les nœuds qui reposent sur la frontière forment une barrière « naturelle ». Si l'on tente de déplacer un nœud sous-surfacique au-delà des nœuds frontières, on créera nécessairement des éléments inversés. Les éléments inversés sont toujours détectés par l'évaluation de la qualité des voisinages élémentaires, ce qui amène les algorithmes de déplacement à proscrire ces positions.

4.2.3 Implémentation du tri de nœuds

Réindexer les nœuds n'implique pas seulement de réordonner les coordonnées des sommets en mémoire ; il faut aussi de mettre à jour les informations de connectivité associées à chacun de ces nœuds. Ces informations comprennent notamment le support topologique, la liste des nœuds adjacents ainsi que la liste des éléments qui partage ce nœud. Les listes de sommets de chaque élément doivent également être mises à jour lors de cette réindexation du tableau de nœuds. C'est pourquoi l'algorithme de tri « tag sort » est plus approprié que d'appliquer « quicksort » directement sur le tableau de nœuds et le tableau de dictionnaires topologique. L'algorithme « tag sort » travaille sur une liste d'indices plutôt que de travailler directement sur la liste des objets à trier. Bien que « quicksort » bénéficie d'une complexité algorithmique de $\mathcal{O}(n * logn)$, n étant le nombre d'éléments à trier, « tag sort » bénéficie de la même complexité algorithmique pour générer le tableau de réindexation, mais est également en mesure de réordonner le tableau de nœuds en exactement n permutations. On réduit donc le nombre de copies en mémoire. De plus, le tableau de réindexation, qui est un résultat intermédiaire du « tag sort », sera utilisé tel quel pour mettre à jour les dictionnaires de connectivité. Le pseudocode 4.1 illustre le fonctionnement de « tag sort » une fois la classification par positions topologiques et par ensembles indépendants des nœuds complétée.

4.3 Déplacement de nœuds sur GPU

Pendant notre courte introduction à l'architecture des GPU, on a vu que la grille de calcul d'un noyau s'étendait sur trois dimensions. De plus, on a vu que les blocs qui composaient cette grille étaient identiques les uns aux autres. Les blocs regroupent quelques fils d'exécution pour leur permettre de travailler en coopération. Ces petits groupes peuvent se synchroniser entre eux et partager de la mémoire de faible latence.

```
Fonction TagSort(Nœuds[], Topologies[]) :
    /* Tri des nœuds par index                                          */
    index = [0, ..., nbNoeuds-1]
    quicksort(index, opérateur< = NoeudPlusPetit(IdA, IdB))

    /* Trouver la nouvelle position de l'ancien nœud i                  */
    position = [nbNoeuds]
    pour i = [0, ..., nbNoeuds-1] faire
        positions[index[i]] = i
    fin

    /* Mise à jour des dictionnaires topologiques                       */
    pour i = [0, ..., nbNoeuds-1] faire
        pour v [0, ..., nbVoisins-1] faire
            Topologies[i].noeudsVoisins[v] = positions[i] ;
        fin
    fin

    /* Mise à jour des tableaux de nœuds et de dictionnaires topologiques
       */
    pour i = [0, ..., nbNoeuds-1] faire
        tant que positions[i] != i faire
            j = positions[i]
            Permuter(Noeuds[i], Noeuds[j])
            Permuter(Topologies[i], Topologies[j])
            Permuter(positions[i], positions[j])
        fin
    fin
fin

Fonction NoeudPlusPetit(IdA, IdB) :
    si PositionTopologiques[IdA] != PositionTopologiques[IdB] alors
        retourner PositionTopologiques[IdA] < PositionTopologiques[IdB]
    sinon
        retourner GroupesIndependants[IdA] < GroupesIndependants[IdB]
    fin
fin
```

Pseudocode 4.1 : L'algorithme « tag sort » pour trier le tableau de nœuds

On appellera espace de parallélisation l'espace que couvrent les fils d'exécution dans la grille de calcul. Pour les algorithmes de déplacement de nœuds, l'espace le plus intuitif à utiliser est l'ensemble des nœuds du maillage. On tient pour acquis, bien que les articles soient rarement explicites à ce sujet, que toutes les recherches présentées à la section 2.7 se limitent à cet espace. L'espace des nœuds est une traduction directe du modèle de parallélisation utilisé sur le CPU, et c'est pourquoi, à ce jour, il semble être le choix par défaut des implémentations GPU. Cependant, comme on le verra à la section 4.3.4, il est possible de paralléliser les algorithmes dans d'autres espaces comme celui des éléments et celui des positions.

4.3.1 Un nœud par fil

C'est l'approche utilisée pour paralléliser les algorithmes *Laplace à ressorts*, *Laplace de qualité*, *Descente du gradient* et *Nelder-Mead*. La taille des blocs n'a aucune importance du point de vue algorithmique. Elle doit être choisie en fonction de la carte graphique utilisée. L'idée est de satisfaire le mieux possible les ordonnanceurs des *Streaming Multiprocessors*. La taille idéale est toujours trouvée de manière empirique.

La traduction des algorithmes depuis le C++ vers GLSL et CUDA est extrêmement simple : il suffit généralement de copier-coller le code, corriger les petites différences syntaxiques et ajouter un en-tête spécifique au langage. En fait, le C++, GLSL et CUDA sont trois langages de programmation basés sur le langage C. C'est pourquoi ils partagent une syntaxe et une manière d'exprimer les algorithmes très similaires. À la limite, ces traductions pourraient être générées automatiquement par un script. L'automatisation de cette traduction faciliterait grandement la maintenance de toutes ces versions.

On observera à l'analyse des résultats que les algorithmes parallélisés de cette manière démontrent de bonnes accélérations. Par contre, si l'on étudie le comportement des noyaux d'un point de vue théorique, on remarque qu'il y a probablement une perte d'efficacité liée à l'hétérogénéité des voisinages élémentaires. Puisqu'il n'y a qu'un pointeur d'instruction par *warp*, on peut s'attendre à un haut taux de divergence pour les voisinages élémentaires composés de différents types d'éléments et de tailles variables. C'est pourquoi uniformiser l'exécution des fils d'un même bloc pourrait être avantageux.

4.3.2 Un nœud par bloc

En vue de réduire la divergence des fils d'exécution, on aimerait homogénéiser la composition des voisinages élémentaires traités dans un même bloc. Il est possible d'y arriver en exploitant adéquatement les espaces de parallélisation. Jusqu'ici, seul l'axe des nœuds a été

considéré. En incorporant l'axe des éléments et l'axe des positions, on se retrouve avec plusieurs fils d'exécution qui traiteront le même voisinage élémentaire. Il est difficile d'obtenir plus d'homogénéité puisque traiter le même voisinage élémentaire implique nécessairement le traitement du même nombre d'éléments du même type.

Cette approche rend l'élaboration d'algorithmes de déplacement de nœuds beaucoup plus spécifique au GPU. Il est difficile de traduire automatiquement ces algorithmes puisqu'il faut dorénavant concevoir des blocs de calcul adaptés à l'espace de parallélisation choisi et assurer la synchronisation des fils à certains points stratégiques. Dans certains cas, on veut également modifier légèrement le comportement de l'algorithme pour bénéficier de ressources inexistantes sur le CPU comme la mémoire partagée.

On inclut également dans cette catégorie les algorithmes qui utilisent plusieurs fils d'exécution pour un nœud sans pour autant empêcher un bloc de traiter plusieurs nœuds à la fois. Les versions multiposition et multiélément de la descente du gradient, présentées à la section 4.3.4, en sont de bons représentants. Ces versions démontrent les mêmes particularités d'implémentation sur GPU que la descente du gradient multiaxe, mais utilisent suffisamment peu de ressources pour admettre plus d'un nœud par bloc.

4.3.3 Grilles de calcul

À chaque exécution d'un noyau, l'intervalle des nœuds à déplacer est envoyé au GPU. Cet intervalle s'exprime par un indice de base ($GroupBase$) qui pointe dans le tableau de nœuds et une taille ($GroupSize$) qui indique combien de nœuds à partir de l'indice de base doivent être déplacés. Puisqu'on a réordonné le tableau de nœuds à la section 4.2.3 en fonction de la classification topologique et des ensembles de nœuds indépendants, les nœuds à déplacer occupent un intervalle contigu dans le tableau de nœuds. Selon les axes de parallélisation choisis pour implémenter les noyaux, un nœud peut être traité par plus d'un fil d'exécution à la fois. Cette distribution est reflétée par la configuration de la grille de calcul.

Le GPU reçoit toujours le travail à effectuer sous forme de grille de calcul. Cette grille tridimensionnelle possède deux niveaux d'organisation : les blocs et les fils. Dans notre cas, les blocs de la grille sont toujours alignés sur l'axe des x. La disposition des fils dépend des axes de parallélisation choisis. Conceptuellement, on réservera l'axe des x pour paralléliser les nœuds. L'axe des y représentera l'axe des éléments et l'axe des z celle des positions.

Lorsqu'un seul fil d'exécution est assigné par nœud, les fils sont tous alignés sur l'axe des x à l'intérieur des blocs. L'index du nœud à déplacer est donné par la position globale du fil d'exécution sur l'axe des x. En GLSL, $gl\_GlobalInvocationID.x$ donne l'indice global

d'un fil sur l'axe des x dans la grille de calcul. L'indice du nœud dans le tableau de nœuds à déplacer est donné par : $vID = GroupBase + gl\_GlobalInvocationID.x$. Dans le code du noyau, on désactive manuellement le fil d'exécution si $gl\_GlobalInvocationID.x >= GroupSize$ pour éviter de traiter des nœuds hors de l'intervalle spécifié. Puisque les fils d'exécution sur le GPU sont toujours invoqués sous forme de blocs et que le nombre de nœuds à traiter n'est généralement par divisible entièrement par la taille des blocs choisie, on aura régulièrement des blocs partiellement remplis. C'est pourquoi il faut empêcher manuellement les fils d'exécution excédentaires de manipuler des nœuds hors de l'intervalle. Ce type d'espace où un nœud est assigné à chaque fils d'exécution de la grille est illustré à la figure 4.5.

Lorsque plus d'un fil d'exécution est assigné par nœud, on utilisera l'axe des y et des z pour disposer les fils à l'intérieur des blocs. Cette fois-ci, tous les fils qui possèdent la même position globale en x dans la grille de calcul traitent le même nœud. C'est-à-dire que l'indice du nœud à déplacer est toujours donné par : $vID = GroupBase + gl\_GlobalInvocationID.x$. Toutefois, les fils ont des rôles différents selon leur position sur l'axe des y et des z. Notamment, on remarquera que les fils en $y = 0$ et $z = 0$ auront régulièrement pour tâche de sélectionner la meilleure position du nœud dans l'ensemble des positions testées (ex : *force brute*).

4.3.4 Lisseurs multiaxes

Quatre nouvelles versions des algorithmes de lissage ont spécifiquement été développées pour le GPU. Ces quatre algorithmes n'ont du sens qu'exécutés sur des processeurs possédant un grand nombre de cœurs. Leur rôle est de proposer une nouvelle approche dans le développement d'algorithmes pour GPU et ainsi inciter les chercheurs en adaptation de maillage à développer des algorithmes spécifiques aux processeurs SIMD plutôt qu'à tenter de porter les algorithmes séquentiels existants.

On commencera par l'implémentation la plus simple : le déplacement de nœuds par force brute. Cet algorithme démontre qu'en misant sur les forces du GPU, il est possible d'obtenir des taux d'accélération bien plus grands qu'en adoptant l'approche traditionnelle d'un fil par

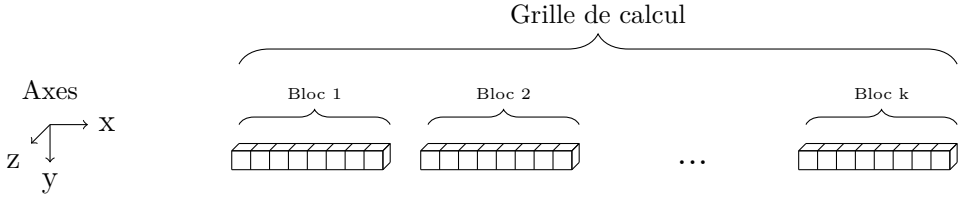


Figure 4.5 Les blocs disposés sur l'axe des x (axe des nœuds) disposent eux-mêmes leurs fils d'exécution sur l'axe des x

nœud. Puis, on tentera d'appliquer le même principe de parallélisation à un algorithme déjà bien connu : la descente du gradient. On étudiera trois combinaisons d'axes de parallélisation afin de déterminer quelle est la division du travail qui permet la plus grande accélération.

Force brute multiposition

Par la force brute, cet algorithme tente de trouver la position idéale d'un nœud en testant une grappe de positions. Cette grappe est distribuée uniformément autour de la position courante du nœud. L'approche de parallélisation est radicalement différente des algorithmes vus précédemment : au lieu d'assigner un seul fil d'exécution, on assigne un bloc entier à chaque nœud. On assigne une position dans la grappe de recherche à chacun des fils du bloc. Les fils doivent évaluer la qualité du voisinage élémentaire en fonction de leur position relative, à la suite de quoi le premier fil du bloc a la responsabilité de déplacer le nœud à l'endroit où la qualité mesurée est la meilleure. La forme de la grappe est la même pour tous les nœuds. On prendra soin de mettre à l'échelle la dimension de la grappe en fonction du rayon du voisinage élémentaire.

En terme d'espace de parallélisation, on voit qu'un bloc entier est alloué à chaque nœud du maillage. Ces blocs sont toujours alignés sur l'axe des x de la grille. Les fils d'exécution, à l'intérieur de chaque bloc, sont quant à eux disposés sur l'axe des z puisqu'ils parallélisent le traitement en fonction des positions de la grappe.

La particularité de cet algorithme réside dans le fait qu'aucun calcul préliminaire n'est nécessaire pour déterminer la direction la plus probable du maximum local. On fait le pari qu'en minimisant les branchements conditionnels et qu'en uniformisant le plus possible les exécutions des fils, on arrivera à maximiser l'efficience de la parallélisation sur GPU. Effectivement, cet algorithme ne produit presque aucune divergence de branche à l'exécution. Même la mesure de qualité du voisinage élémentaire est parfaitement uniforme, car, contrairement aux algorithmes précédents, les fils d'exécution d'un même bloc mesurent la qualité du même voisinage élémentaire. Cela garantit l'égalité du nombre et du type des éléments à évaluer.

Le point faible du déplacement par tests simultanés est qu'il doit tester plus de positions que les autres algorithmes pour offrir la même précision. Puisqu'il est conseillé de définir des tailles de bloc divisibles par 32 en CUDA (taille d'un *warp*, granularité minimale pour les cartes NVIDIA), nous avons déterminé que des blocs de 64 fils d'exécution seraient optimaux. Ceci nous laisse donc 64 positions à tester autour du nœud. La manière la plus simple de disposer ces 64 positions est de former une grille uniforme de taille 4x4x4 centrée sur le nœud. Cette disposition offre une convergence comparable aux algorithmes de descente du gradient et Nelder-Mead.

Notons que les algorithmes classiques de déplacements de nœuds testent bien moins de 64 positions par cycle. Cet excès d'opérations réduit l'efficience de l'algorithme et lui donne un certain retard face aux algorithmes classiques. On constatera toutefois au chapitre 5 que cet algorithme démontre une bien meilleure accélération qu'eux sur le GPU, ce qui justifie sa compétitivité.

Le noyau de calcul est divisé en trois sections : chargement en mémoire, tests simultanés des positions et sélection du meilleur candidat. D'abord, l'ensemble des éléments du voisinage élémentaire est chargé de la mémoire principale et mis en « cache » dans la mémoire partagée. Puisque la mémoire partagée réside dans les mêmes registres que la mémoire cache, elle est bien plus rapide que la mémoire principale tout en restant accessible à l'ensemble des fils d'exécution d'un même bloc. Chaque fil d'exécution s'occupe de charger un des éléments du voisinage élémentaire ce qui permet de charger le tout en une seule opération. Puis, chaque fils d'exécution évalue la qualité du voisinage élémentaire en fonction de sa position dans la grappe. Finalement, le premier fil d'exécution du groupe s'occupe de trouver la meilleure position dans la grappe selon les qualités mesurées et déplace le nœud à cet endroit. La configuration de la grille de calcul utilisée par l'algorithme de force brute est présentée à la figure 4.6.

Descente du gradient multiaxe

Les trois prochaines versions de l'algorithme « descente du gradient » reprennent la même implémentation CPU, mais tentent d'accélérer l'algorithme sur le GPU à l'aide de différents espaces de parallélisation. On commencera par le niveau de parallélisme le plus élevé et l'on tentera de déterminer par les versions suivantes quel est l'espace ayant le plus grand impact sur la rapidité de l'algorithme. Dans la version multiaxe, c'est-à-dire à la fois multiposition et multiélément, le travail est distribué sur des blocs en deux dimensions. Chaque bloc de calcul s'occupe uniquement d'un nœud. Dans un bloc, les fils se partagent les tâches nécessaires

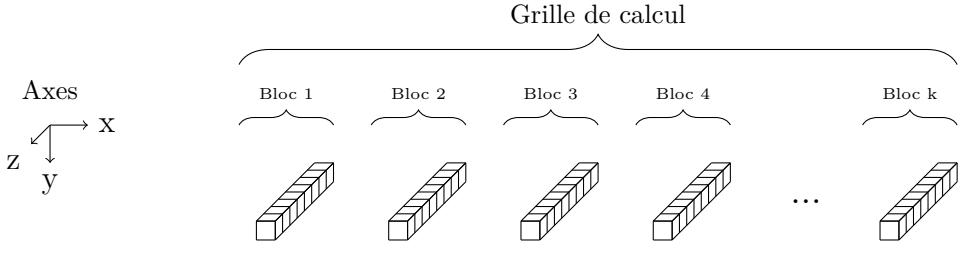


Figure 4.6 Les blocs disposés sur l'axe des x (axe des nœuds) disposent leurs fils d'exécution sur l'axe des z (axe des positions)

au déplacement du nœud en se répartissant les éléments de son voisinage élémentaire et les positions intermédiaires que doit occuper le nœud.

La première dimension du traitement parallèle se partage les éléments du voisinage élémentaire du nœud déplacé. Les fils d'exécution calculent la qualité des éléments qui leur sont assignés puis soumettent leurs résultats dans une variable d'accumulation partagée. Les fils responsables des premiers éléments du voisinage élémentaire, c.-à-d. d'indice $(x, y = 0, z)$, finalisent ensuite la qualité des voisinages élémentaires. Pour terminer, les fils d'exécution d'indice $(x, y = 0, z = 0)$ utilisent les qualités calculées pour définir la droite de recherche à l'aide du gradient puis sélectionnent la position optimale du nœud sur la droite de recherche.

La deuxième dimension s'occupe des positions à tester simultanément. On notera qu'il y a deux catégories de positions dans l'algorithme du gradient. D'abord, on voudra construire le gradient de la fonction coût à l'aide d'une différence centrée en six points. Ensuite, pour la recherche linéaire, on évaluera la qualité du voisinage élémentaire en huit positions sur la droite de recherche.

Une certaine coordination des fils d'exécution est nécessaire pour offrir un tel niveau de parallélisme. Sept barrières de synchronisation assurent la réception des résultats avant de poursuivre l'exécution. La première est placée juste après la section d'initialisation où les éléments du voisinage élémentaire sont mis en mémoire partagée. Puis, les fils sont synchronisés à trois endroits pour chaque catégorie de positions : on les synchronise après l'évaluation des éléments, après la mis en commun des qualités des voisinages élémentaires et après avoir utilisé les qualités. Autrement dit, les dernières barrières sont placées après le calcul du gradient et après la recherche linéaire.

L'organisation du travail à l'intérieur des blocs de calcul est illustrée à la figure 4.7. Puisqu'un grand nombre de fils d'exécution est nécessaire pour traiter un seul nœud, un bloc entier est assigné à chaque nœud. En fait, on se rappellera qu'un bloc peut contenir un maximum de 1024 fils d'exécution. On doit premièrement fixer la taille du bloc dans l'axe des positions à 8. Ce nombre correspond au nombre d'échantillons que l'on souhaite tester le long de la droite de recherche et couvre également les 6 échantillons nécessaires pour estimer le gradient. Dans l'axe des éléments, on souhaiterait bien sûr assigner un fil d'exécution à chaque élément du voisinage élémentaire, mais on manquera malheureusement de ressources. La taille théorique maximale est $1024/8 = 128$, ce qui dépasse largement la quantité totale d'éléments des voisinages élémentaires les plus volumineux (généralement autour de 70 éléments). Mais, les *Streaming Multiprocessors* des cartes graphiques ne possèdent souvent pas suffisamment de registres pour pouvoir exécuter plus de 256 fils d'exécution de cet algorithme. Cela nous laisse donc $256/8 = 32$ fils par position pour traiter tous les éléments d'un voisinage élémentaire.

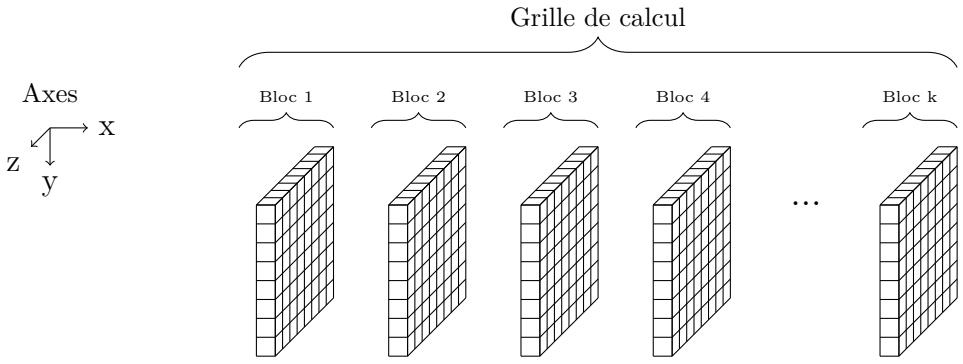


Figure 4.7 Les blocs disposés sur l'axe des x (axe des nœuds) disposent leurs fils d'exécution sur l'axe des y (axe des éléments) et sur l'axe des z (axe des positions)

Chaque fil doit donc évaluer la qualité de deux à trois éléments.

La parallélisation de l'algorithme à la fois dans l'axe des positions et des éléments offrent de bonnes performances, mais consomme une grande quantité de ressources mémoires. De plus, on aimerait déterminer dans quelle proportion les deux axes de parallélisation accélèrent l'algorithme. C'est pourquoi on réécrira l'algorithme en supprimant alternativement l'un et l'autre des axes.

Descente du gradient multiposition

On commence la dissection de l'algorithme en isolant l'effet du parallélisme par position. C'est-à-dire qu'un fil d'exécution mesurera la qualité de tous les éléments du voisinage élémentaire en fonction d'une seule position. Encore une fois, on choisira 8 fils d'exécution par nœuds pour tester les groupes de positions.

Cela nous laisse maintenant beaucoup de fils libres par bloc. Puisque la taille d'un *warp* sur les cartes NVIDIA est de 32 fils, on traitera $32/8 = 4$ nœuds au minimum en simultané par bloc, tel qu'illustré à la figure 4.8. Mais, puisqu'il n'est pas possible de réserver plus d'espace dans la mémoire partagée, il n'est plus possible de précharger les éléments en mémoire partagée si l'on veut assigner plus d'un voisinage élémentaire par bloc. On se privera donc entièrement de cette optimisation et on lira les éléments directement en mémoire principale.

Cette version de la descente du gradient garde la même structure algorithmique que la version multiaxe. La seule différence est que chaque fil parcourt tous les éléments du voisinage élémentaire. Ainsi, les fils peuvent calculer directement la qualité du voisinage élémentaire sans l'aide de barrière de synchronisation. Cette version compte donc deux barrières de synchronisation de moins que la version multiaxe.

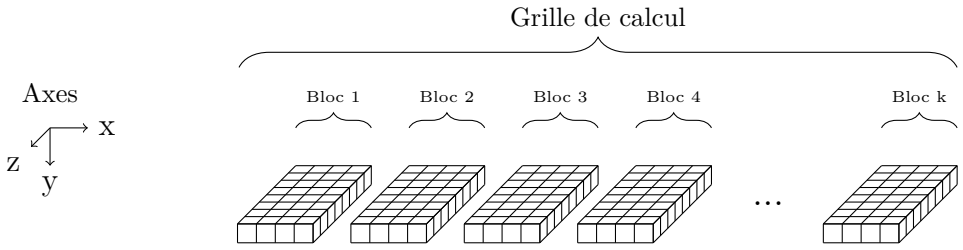


Figure 4.8 Les blocs disposés sur l'axe des x (axe des nœuds) disposent leurs fils d'exécution sur l'axe des x (axe des nœuds) et sur l'axe des z (axe des positions)

Descente du gradient multiélément

On dissèque ici l'algorithme dans l'autre direction de l'espace de parallélisation en isolant le parallélisme par élément. On attribue un petit lot d'éléments à chaque fil. Les fils testeront l'un après l'autre toutes les positions sur leur lot d'éléments. On gardera 32 fils d'exécution par voisinage élémentaire, ce qui nous laisse de la place pour traiter plus d'un voisinage élémentaire par bloc. La configuration de la grille de calcul est donnée à la figure 4.9.

Contrairement aux versions multiaxe et multiposition, cet algorithme comporte deux courtes boucles imbriquées. De plus, cet algorithme ne peut pas se débarrasser de la même barrière de synchronisation que la version multiposition puisque les fils d'exécution ne traitent qu'une portion du voisinage élémentaire. Il faut donc encore attendre que tous les fils aient évalué leurs éléments avant de calculer la qualité du voisinage élémentaire dans son ensemble.

4.4 Structures de données GPU

Maintenant que l'on sait comment traduire nos algorithmes de déplacement de nœuds du CPU au GPU, il faut définir les moyens de communication entre les deux processeurs. Le

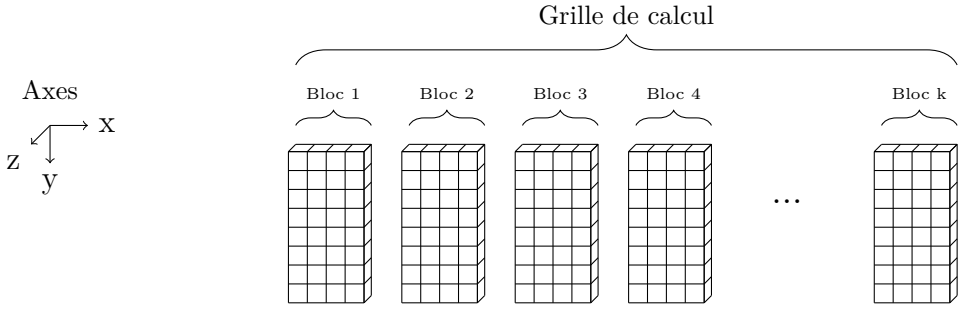


Figure 4.9 Les blocs disposés sur l'axe des x (axe des nœuds) disposent leurs fils d'exécution sur l'axe des x (axe des nœuds) et sur l'axe des y (axe des éléments)

maillage et sa métrique spécifiée ont déjà une représentation bien définie sur le CPU. Puisque les deux processeurs possèdent des espaces mémoire distincts, on doit traduire ces structures de données dans un format compatible avec le GPU et copier le résultat dans son espace mémoire. Une fois les déplacements de nœuds complétés, les données modifiées doivent être copiées vers le CPU et traduites à nouveau dans le format d'origine. Quelques classes ont été développées pour encapsuler ce comportement.

En tout temps, des instances C++ du maillage et de la métrique sont actives dans le logiciel d'adaptation. On crée de nouvelles instances GLSL et CUDA du maillage et de la métrique qu'au moment de lancer l'implémentation GPU d'un algorithme. Lorsque le processus d'adaptation termine, l'instance C++ du maillage est mise à jour et les instances GPU sont détruites. Cette façon de faire minimise l'utilisation de la mémoire GPU et permet d'inclure les temps dédiés à la gestion de la mémoire (traduction et copie) dans les bancs de tests. Aucune opération de traduction ou de copie n'est nécessaire pour les algorithmes du CPU. Il est donc important de mesurer l'impact de ces opérations sur les performances des algorithmes GPU afin de présenter des comparaisons équitables.

Dans cette section, on présentera les classes et structures de données auxiliaires développées pour traduire les données dans un format compatible avec le GPU et synchroniser les différentes instances du maillage et de la métrique spécifiée. Chaque langage GPU étudié possède sa propre instance du maillage et de la métrique. Heureusement, les formats de données de GLSL et CUDA sont compatibles, ce qui nous permet d'avoir une seule fonction de traduction pour chaque structure de données. Toutefois, bien que les programmes des différents langages résident sur le même coprocesseur, leur espace mémoire n'est pas partagé. On doit utiliser les fonctions de transfert de mémoire spécifiques à chaque langage lorsqu'on veut communiquer avec ces algorithmes.

4.4.1 Traduction du maillage

Le format du maillage sur le GPU est similaire à celui que l'on a conçu pour le CPU. On s'est déjà assuré de n'utiliser que des structures et des tableaux de type C, ce qui nous facilite grandement le travail. On doit toutefois faire attention au format des nombres. Par exemple, un *int* C++ n'est pas nécessairement stocké sur le même nombre d'octets qu'un *int* GLSL. De plus, puisque les cartes graphiques testées sont beaucoup moins performantes en double précision qu'en simple, on représentera la position des nœuds par des nombres à virgule flottante en simple précision sur le GPU.

Cette différence de performance est due au nombre inférieur d'unités arithmétiques double précision par rapport aux unités simple précision pour les opérations en virgule flottante. La

GEFORCE GTX 780 Ti® possède 24 fois moins d'unité double précision que d'unités simple précision. La raison est que cette carte est destinée au marché des jeux vidéos. Certaines cartes comme les Quadro et les Tesla proposent des ratios de performances de l'ordre de 1/3 entre double et simple précision, mais leur prix est dramatiquement plus élevé. De plus, l'occasion d'utiliser de telles cartes pour produire les résultats de la présente recherche ne s'est pas présentée.

La seule structure qui demande un peu plus de traitement pour être traduite est le tableau de dictionnaires topologiques. Cette structure contient entre autres le support topologique du nœud associé, qui sera tout simplement ignoré puisque les frontières ne sont pas prises en compte sur le GPU. Un dictionnaire contient également deux tableaux : un pour les nœuds voisins et un autre pour les éléments du voisinage élémentaire. Puisque l'allocation dynamique de mémoire n'est pas possible sur le GPU, et que les dictionnaires topologiques sont de tailles variables, on traduira le tableau de dictionnaires topologiques en trois tableaux de tailles fixes : un tableau de références topologiques, un tableau de nœuds voisins et un tableau de voisinages élémentaires. Les références topologiques sont de petites structures de tailles fixes qui indiquent où trouver les nœuds voisins et les éléments des voisinages élémentaires d'un nœud donné dans les deux autres tableaux. Le format pour GPU des dictionnaires topologiques est illustré à la figure 4.10.

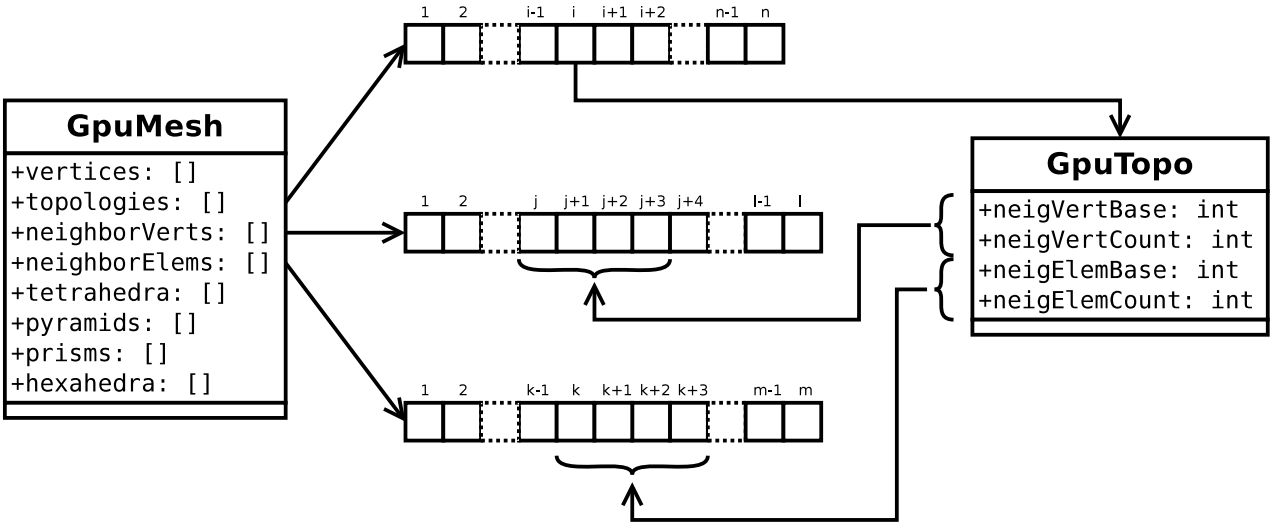


Figure 4.10 Les dictionnaires topologiques pointent vers une sous-liste de nœuds voisins (neighborVerts) et une sous-liste d'éléments voisins (neighborElems)

4.4.2 Traduction de la métrique

La traduction de la métrique dépend du type de représentation choisie. Dans le cas des métriques uniforme et analytique, seul le facteur de mise à l'échelle est communiqué. La métrique analytique est fournie au programme sous la forme d'une procédure. Elle est directement écrite dans le langage du programme (GLSL ou CUDA) dans un fichier source.

Même si l'échantillonnage de la métrique par recherche locale n'est pas tout à fait fonctionnel sur le GPU, celle-ci a quand même été implémentée. Le maillage de fond est une version simplifiée du maillage original. On y retrouve un tableau de nœud, un tableau de tenseurs métriques (un tenseur par nœud) et un tableau de tétraèdres. Les tétraèdres du maillage de fond contiennent la liste de leurs quatre sommets, ainsi que la liste de leurs quatre éléments voisins. Le $i^{\text{ème}}$ tétraèdre voisin est l'élément rejoint si l'on traverse la face opposée au sommet i du tétraèdre courant. Les trois tableaux du maillage de fond peuvent directement être envoyés sur le GPU sans modification des structures de données. Il suffira de traduire les nombres entiers et à virgule flottante dans le bon format.

La traduction de la métrique sous forme de texture 3D demande un traitement particulier. D'abord, on se rappellera que les textures possèdent un maximum de 4 composantes par texel. Les tenseurs métriques possèdent 9 composantes, dont 6 ne sont pas redondantes. On divisera donc le tenseur en deux vecteurs de 3 composantes. Chaque vecteur sera stocké dans une texture distincte. La division du tenseur est illustrée à la figure 4.1. L'échantillonnage des deux textures est le réassemblage du tenseur métrique sont donnés au listage 4.11. On prendra soin d'activer l'interpolation linéaire des textures pour diminuer l'erreur d'interpolation de la métrique. Au moment d'échantillonner la métrique, on échantillonnera les deux textures exactement à la même position. On obtiendra ainsi deux vecteurs de composantes qui nous permettront de réassembler le tenseur métrique complet.

4.4.3 Transferts de mémoire

Une fois les structures de données du maillage et de la métrique traduites, il faut allouer l'espace mémoire nécessaire pour les accueillir sur le GPU. On allouera un tampon par tableau de données. Au moment d'invoquer un algorithme de déplacement de nœuds, on copiera le contenu des traductions dans les tampons mémoires. Pour les implémentations GLSL, on fera appel à *glBufferData*. Pour les implémentations CUDA, on fera appel à *cudaMemCpy*. Une fois toutes les itérations de l'algorithme complétées, on libérera la mémoire des tampons.

Puisque les algorithmes de déplacement de nœuds GPU dépendent du CPU pour déplacer les nœuds frontières, la position des nœuds frontières et des nœuds sous-surfaciques sont

```
uniform sampler3D TextureA;
uniform sampler3D TextureB;
uniform mat4 TexTransform;

mat3 metricAt(in vec3 position)
{
    vec3 coor = vec3(TexTransform * vec4(position, 1.0));

    vec3 vecA = texture(TextureA, coor).xyz;
    vec3 vecB = texture(TextureB, coor).xyz;

    mat3 metric = mat3(
        vecA.x, vecA.y, vecA.z,
        vecA.y, vecB.x, vecB.y,
        vecA.z, vecB.y, vecB.z);

    return metric;
}
```

Figure 4.11 Échantillonnage des textures métriques et réassemblage du tenseur

constamment copiés d'un processeur à l'autre pendant le processus de lissage. Ces transferts de mémoire représentent la plus grande part du temps consacré à la communication entre processeurs. L'ordonnancement de ces copies est présenté à la section qui suit.

4.4.4 Ordonnancement des copies mémoires

Maintenant que l'on connaît la structure du tableau de nœuds en mémoire et la traduction des structures de données pour le GPU, il est possible de définir la fonction pilote des algorithmes de lissage coopératifs CPU-GPU. Pour chacun des ensembles de nœuds indépendants, en plus de lancer les fils d'exécution CPU pour traiter les nœuds frontières, le fil d'exécution principal s'occupe de lancer le noyau de calcul sur le GPU pour traiter les nœuds sous-surfaciques et intérieurs. Une fois l'exécution du noyau de calcul terminé, on copie les nœuds sous-surfaciques du GPU vers le CPU. Puis on attend que tous les fils d'exécution CPU aient terminé de traiter les nœuds frontières avant de copier leur nouvelle position vers le GPU. Une fois cette copie terminée, le pilote peut passer au prochain ensemble de nœuds indépendants jusqu'à ce que le processus de lissage ait convergé. Ce n'est qu'une fois le processus terminé que les nœuds intérieurs doivent être copiés vers le CPU. L'algorithme 4.2 illustre l'ordre des copies mémoires. Le code source des quatre pilotes d'algorithme sont donnés à l'annexe B.

```
Copier tétraèdres vers GPU
Copier pyramides vers GPU
Copier prismes vers GPU
Copier hexaèdres vers GPU

Copier nœuds vers GPU
Copier dictionnaires topologiques vers GPU

tant que convergence pas atteinte faire
    pour chaque ensemble E de nœuds indépendants faire

        Lancer lissage GPU des nœuds sous-surfaciques et intérieurs de E

        pour chaque thread T du thread pool faire
            Lancer lissage CPU des nœuds frontières de E délégués à T
        fin

        Attendre fin du lissage GPU
        Copier nœuds sous-surfaciques de E vers le CPU

        Synchroniser avec tous les threads
        Copier nœuds frontières de E vers le GPU
    fin
fin

Copier nœuds intérieurs vers CPU
Libérer toute la mémoire du GPU

// Le maillage lissé est disponible sur CPU
```

Pseudocode 4.2 : Fonction pilote des algorithmes de lissage sur GPU

4.4.5 Vue d'ensemble du déplacement de nœuds sur GPU

L'ordonnancement des copies mémoires entre le CPU et GPU était le dernier bloc qui manquait pour couvrir l'infrastructure du logiciel d'adaptation. On sait maintenant sous quelle forme est stocké le maillage et comment la fonction métrique est échantillonnée sur le GPU. On sait également de quelle manière le CPU et le GPU communiquent et se synchronisent. Afin de comparer les implémentations équitablement, on inclura toujours les temps dédiés à la copie des structures de données d'un processeur à l'autre dans les temps de total de lissage. C'est-à-dire qu'à la fin de toute exécution, le maillage adapté est prêt à être utilisé depuis la mémoire principale du CPU et toute la mémoire du GPU est libérée. On estime que c'est l'état final souhaité dans un adaptateur puisque l'étape suivante est soit d'exécuter des opérations topologiques sur le maillage ou d'écrire celui-ci sur le disque.

CHAPITRE 5 ANALYSE DES RÉSULTATS

Développer un logiciel d'adaptation de maillage pour le GPU comporte plusieurs difficultés. Il y a d'abord l'élaboration d'algorithmes de déplacement de nœuds appropriés au modèle d'exécution SIMD, mais également le stockage et l'échantillonnage de la fonction métrique. Dans les chapitres précédents, plusieurs algorithmes de déplacement de nœuds et plusieurs techniques d'échantillonnage ont été présentés. Dans le présent chapitre, on évalue leur valeur respective en fonction du type de processeur ciblé. Ultimement, on aimerait assembler les pipelines d'adaptation optimaux de chaque processeur pour les comparer du point de vue de l'efficacité et de la rapidité.

5.1 Processus de comparaison

Le logiciel d'adaptation de maillage développé dans le cadre de cette recherche répartit les tâches en plusieurs modules. L'objectif principal de ce logiciel est de nous aider à déterminer si l'utilisation d'une carte graphique permettrait d'accélérer l'ensemble du processus d'adaptation. Pour y arriver, on portera notre attention sur les deux modules ayant le plus grand impact sur le temps de calcul total : l'échantillonnage de la métrique et le lissage par déplacement de nœuds. Ultimement, on aimerait pouvoir spécifier quels sont les pipelines d'adaptation optimaux pour chaque processeur : un pour le CPU et un deuxième pour le GPU.

Les premières observations porteront sur les performances et les niveaux de précisions des techniques d'échantillonnage. Ces observations nous permettront de choisir les techniques d'échantillonnage les plus pertinentes à mettre en place pour les pipelines d'adaptation CPU et GPU. Ensuite, on se penchera sur les performances des algorithmes de lissage par déplacement de nœuds. On étudiera les gains de qualité obtenus et la rapidité des algorithmes en fonction du type d'implémentation exécuté. Par « implémentation », on entend l'une des quatre formes suivantes : (1) séquentielle en C++ sur CPU, (2) parallèle en C++ sur CPU, (3) en GLSL sur GPU et (4) en CUDA sur GPU. Tous les algorithmes possèdent une implémentation séquentielle et parallèle sur le CPU, mais certains d'entre eux possèdent plus d'une implémentation en GLSL et CUDA. Ces implémentations GPU supplémentaires font appel à différentes combinaisons des espaces de parallélisation.

Une fois les pipelines optimaux choisis et mis en place pour le CPU et le GPU, on comparera leur comportement sur deux configurations matérielles. En premier lieu, on étudiera la crois-

sance du temps de calcul des pipelines en fonction de la taille des maillages adaptés. Ensuite, on testera les pipelines sur des cas réels de mécanique des fluides numérique.

5.2 Environnements de test

La rapidité et l'efficacité des pipelines d'adaptation dépendent de plusieurs variables, notamment de la configuration matérielle utilisée, des cas de tests choisis et des critères de sélections. On ne tente pas ici de couvrir un grand nombre de combinaisons, car cela nuirait à la présentation des résultats. On a donc choisi les configurations matérielles typiquement utilisées en ingénierie, quelques cas de tests synthétiques simples et quelques cas réels pour les tests finaux. Quant aux critères de sélection, puisqu'il n'existe pas de seuil de convergence absolu, on s'appuiera sur des nombres d'itérations fixes pour tirer nos conclusions.

5.2.1 Configurations matérielles

Le principal avantage d'utiliser une carte graphique comme plateforme d'accélération est que ce type de processeur est largement répandu et peu coûteux. Cela s'explique par le vaste marché dont les cartes graphiques disposent dans le domaine des jeux vidéos. De plus, une bonne carte graphique est nécessaire à tout ingénieur ou chercheur actif dans le domaine de la CAO. Par conséquent, on peut s'attendre à retrouver des cartes graphiques dans plusieurs types d'ordinateurs destinés aux ingénieurs : que ce soit les superordinateurs, les ordinateurs de bureau ou les ordinateurs portables. Afin d'analyser la pertinence des GPU en adaptation de maillage, il semble pertinent de tester différents types de configurations matérielles. On peut toujours entrevoir un jour où il sera concevable d'effectuer des simulations en mécaniques des fluides numériques sur des ordinateurs personnels qui étaient autrefois réservées aux superordinateurs à cause de leur temps de calcul prohibitifs.

Deux configurations matérielles seront testées. L'opportunité d'accéder à un superordinateur ne s'est pas présentée, d'où l'utilisation exclusive d'un ordinateur de bureau et d'un ordinateur portable. La première configuration s'approche du poste de travail typique de l'ingénieur. Il est équipé d'un processeur à 4 cœurs avec hyperthread et d'une carte graphique haut de gamme pour jeu vidéo. La deuxième configuration se classe plutôt comme un ordinateur portable de moyenne gamme, équipé d'une carte graphique également de moyenne gamme. Ces deux configurations sont équipées de cartes graphiques NVIDIA suffisamment récentes pour pouvoir exécuter les implémentations CUDA du logiciel d'adaptation. Puisque CUDA est une technologie propriétaire de NVIDIA, les cartes graphiques produites par ATI ne peuvent exécuter les implémentations développées sous cette forme. D'où l'intérêt des implémentations

GLSL qui sont plus portables. Le détail des configurations est présenté au tableau 5.1

La configuration matérielle par défaut, utilisée pour produire tous les résultats, sauf indication contraire, est la première : le Intel® Core™ i7-6700K équipé d'une carte graphique NVIDIA GeForce GTX 780 Ti. L'autre configuration sera utilisée à la fin de ce chapitre pour évaluer l'extensibilité de la solution proposée.

5.2.2 Cas de test

Les cas de test proposés pour évaluer les pipelines d'adaptation sont majoritairement synthétiques. Les choix des pipelines optimaux pour le CPU et le GPU sont faits à partir de maillages en forme de sphère et de cube. La sphère ne possède aucun sommet topologique, aucune arête topologique et une seule surface topologique. Le cube possède 8 sommets topologiques, 12 arêtes topologiques et 6 surfaces topologiques. La représentation de sa frontière est un peu plus complexe que celle de la sphère, mais également plus représentative des cas réels. Entre autres, lors des opérations topologiques, il faut respecter l'appartenance des nœuds à leur support topologique. La sphère est toujours maillée à l'aide de tétraèdres tandis que le cube sera tantôt maillé à l'aide de tétraèdres, tantôt à l'aide d'hexaèdres.

La métrique spécifiée fournie avec les cas de test synthétiques est définie analytiquement et peut être exprimée mathématiquement par l'équation 5.3. D'abord, un facteur d'échelle K permet de contracter l'espace. Par exemple, en utilisant un facteur d'échelle $K = 5$, les arêtes qui mesurent 1 unité dans l'espace euclidien mesurent 5 unités dans la métrique. Puis, un facteur de compression A permet de changer le rapport d'aspect des éléments dans la direction du vecteur $\vec{u}$. Le facteur de compression module une fonction sinusoïdale. Au minimum de la fonction, le ratio est égal à 1 et au maximum il est égal à A. La fonction sinusoïdale simule des chocs dans la solution qui seraient perpendiculaires au vecteur $\vec{u}$. Plus A est élevé, plus le rapport d'aspect augmente et plus les chocs sont concentrés sur les plans qui supportent les maximums de la fonction.

Tableau 5.1 Spécifications des configurations matérielles

Spécifications	Desktop NVIDIA	Laptop NVIDIA
Type	Poste de travail	Portable
OS	Ubuntu 16.04.1 64-bit	Ubuntu 16.04.1 64-bit
CPU	Intel® Core™ i7-6700K	Intel® Core™ i7-3610QM
Cœurs	8 × 4.00GHz	8 × 2.30GHz
GPU	GeForce GTX 780 Ti	GeForce GT 650M
Pilotes	NVIDIA 367.57	NVIDIA 381.22

Le vecteur $\vec{u}$ est choisi pour éviter d'aligner les chocs avec les cellules des textures. Si les chocs étaient alignés simplement avec l'axe des x, le crénelage de la métrique dans la texture serait minimisé, ce qui pourrait cacher des défauts de précision. Le vecteur $\vec{u}$ est obtenu depuis le vecteur $(1, 0, 0)$ à l'aide de deux rotations : une première de $-\pi/4$ autour de l'axe des y suivit d'une rotation de $\pi/4$ autour de l'axe des z. L'application successive de ces deux rotations est donnée par la matrice R :

$$R = \begin{bmatrix} cos(\pi/4) & -sin(\pi/4) & 0 \\ sin(\pi/4) & cos(\pi/4) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos(-\pi/4) & 0 & sin(-\pi/4) \\ 0 & 1 & 0 \\ -sin(-\pi/4) & 0 & cos(-\pi/4) \end{bmatrix} \tag{5.1}$$

Les ondes de choc simulées par la métrique suivront la position de l'échantillon $\vec{p}$ sur l'axe u :

$$\vec{p}_u = (1, 0, 0) \cdot R\vec{p} \tag{5.2}$$

À partir de la position de l'échantillon sur l'axe u, on construit une métrique M dans la base orthonormée $(\vec{u}, \vec{v}, \vec{w})$. Les valeurs propres de M associées aux axes v et w sont données directement par le carré du facteur d'échelle $\lambda_v = \lambda_w = K^2$. La valeur propre associée à l'axe u est obtenue à l'aide de la fonction sinusoïdale de choc illustrée à la figure 5.1.

$$M = \begin{bmatrix} K_u^2 & 0 & 0 \\ 0 & K^2 & 0 \\ 0 & 0 & K^2 \end{bmatrix}, K_u = K \times A^{((1-cos(2\pi\vec{p}_u))/2)^A} \tag{5.3}$$

Ce qui nous amène à la métrique spécifiée finale M_S donnée par l'application de la rotation R sur le tenseur M :

$$M_S = R^{-1} \cdot M \cdot R \tag{5.4}$$

Un exemple de maillage adapté à l'aide de la métrique M_S est donné à la figure 5.2.

Pour s'assurer que les résultats tirés des tests synthétiques restent valides dans la pratique, on teste également les pipelines d'adaptation sur un maillage réel tiré de travaux d'ingénierie. Le cas de test représente un jet dans une boîte rectangulaire maillée à l'aide de 32 000 nœuds et de 142 595 tétraèdres. Sa métrique spécifiée est calculée à partir d'une estimation de l'erreur de discrétisation *a posteriori*.

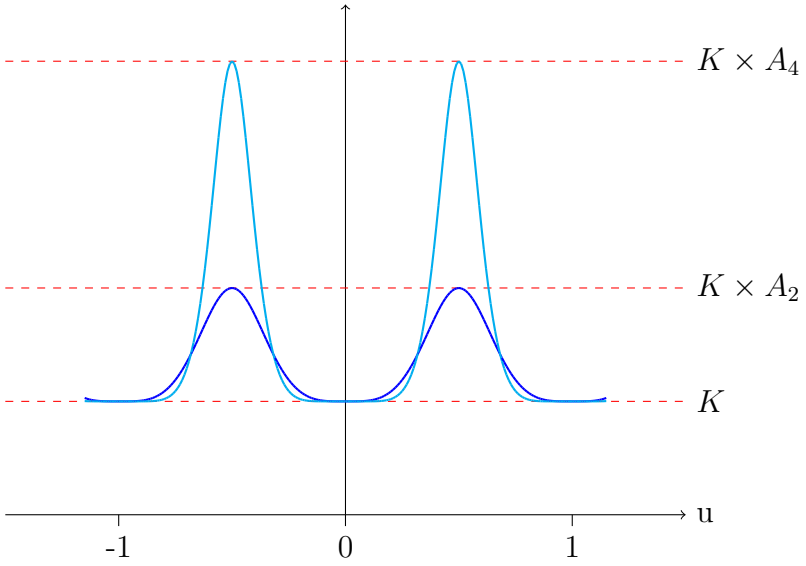


Figure 5.1 Graphique de la fonction métrique : $K \times A^{((1-cos(2\pi\vec{p}_u))/2)^A}$

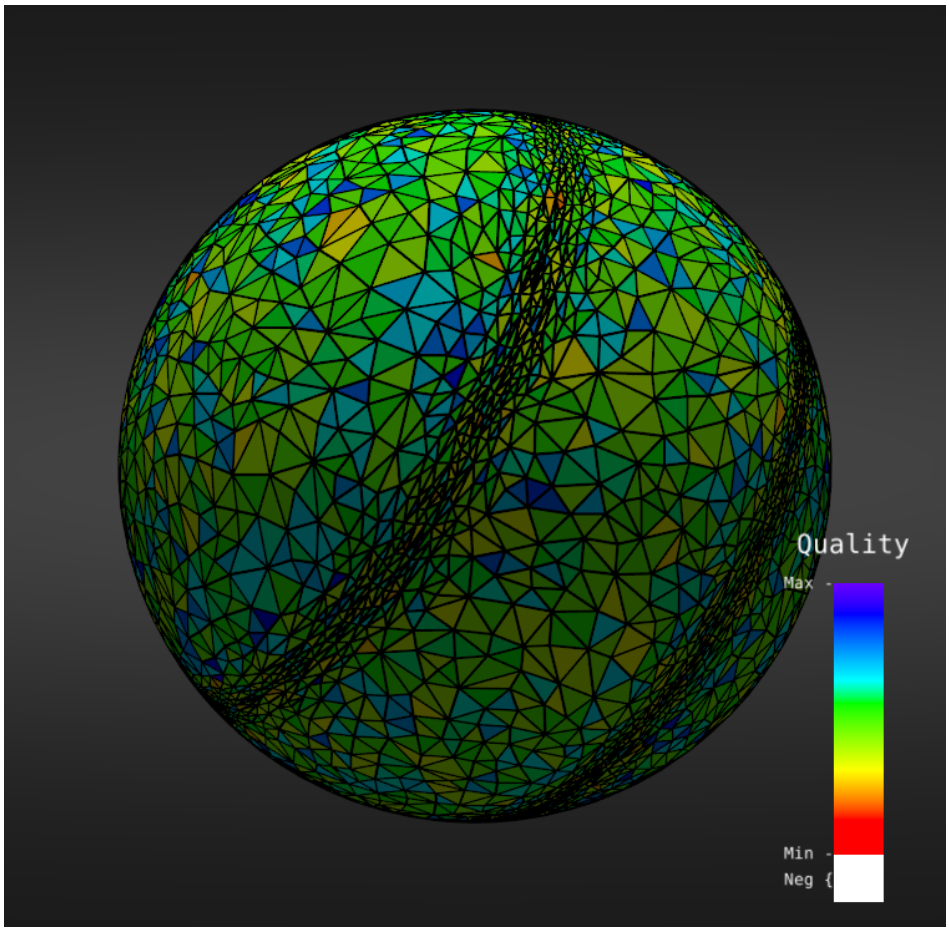


Figure 5.2 Maillage d'une sphère adaptée à la métrique sinusoïdale ($K = 12$, $A = 8$)

Toutes les qualités présentées dans ce chapitre sont données par la mesure de conformité à la métrique $\mathscr{C}$. Puisqu'elle est la seule mesure capable d'évaluer à la fois la forme et la taille des éléments par rapport à la fonction métrique, elle représente le meilleur choix de mesure en adaptation de maillage. À partir d'ici, et jusqu'à la fin du présent chapitre, les termes qualité et conformité à la métrique seront utilisés de manière interchangeable.

5.2.3 Critères de sélection

Il n'est pas nécessaire que les deux configurations optimales des pipelines d'adaptation sur le CPU et le GPU utilisent la même technique d'échantillonnage et le même algorithme de déplacement de nœuds, tant que les gains en qualité qu'elles fournissent sont équivalents. Les choix des configurations se feront indépendamment l'un de l'autre. C'est-à-dire que lorsque l'on se penchera sur la meilleure configuration pour le GPU, on fera abstraction du choix réalisé pour le CPU et vice-versa.

Pour sélectionner les configurations optimales, nous nous baserons sur deux critères : l'efficacité et la rapidité. L'efficacité sera mesurée à l'aide des implémentations parallèles sur CPU. On observera que, des deux valeurs proposées pour mesurer la qualité d'un maillage, seule la moyenne harmonique des qualités des éléments offre une bonne base de comparaison, car la qualité du pire élément ne croit généralement pas de manière monotone tout au long du processus d'adaptation. Cela est dû à la fonction de coût choisie pour optimiser les voisinages élémentaires : la moyenne harmonique. Le critère de rapidité est quant à lui mesuré pour toutes les implémentations, en secondes ou en millisecondes. De plus, on rencontrera régulièrement la rapidité exprimée sous forme de coefficient d'accélération face aux implémentations séquentielles.

On remarquera qu'en général, ces deux critères s'opposent. C'est-à-dire que, souvent, plus un algorithme est rapide, plus son gain en qualité est petit. Pour déterminer les configurations optimales, un poids subjectif devra être accordé à chacun des critères. Une manière simple de départager les configurations pourrait être de calculer la vitesse du gain en qualité :

$$gain_{harmonique}/s = \frac{\alpha_{harmonique}(\mathscr{M}_{10}) - \alpha_{harmonique}(\mathscr{M}_0)}{t} \tag{5.5}$$

où $\mathscr{M}_i$ est le maillage après i itérations. Cette équation donne une idée de cette vitesse. Toutefois, elle fait abstraction des phénomènes internes aux algorithmes de déplacement de nœuds. D'abord, les gains de chaque itération diminuent au fil du processus de déplacement de nœuds. Puis, plus certains algorithmes convergent, plus ils présentent des itérations courtes. Ces phénomènes sont illustrés à la figure 5.3 par les cinq premières itérations de Nelder-

Mead exécutées pour adapter le maillage de la figure 5.2. Une mesure précise et uniforme de la vitesse du gain est donc difficile à obtenir. Dans notre cas, on se basera simplement sur le temps que prennent les implémentations à compléter un nombre fixe d'itérations. On fera de même pour comparer les efficacités qui seront comparées sur la base d'un nombre fixe d'itérations.

Bien que rapidité et efficacité s'opposent généralement, il n'est pas cependant impossible de satisfaire les deux à la fois. Comme le propose Lo (2015), rien n'empêche d'utiliser une succession d'algorithmes de déplacement de nœuds. Les plus rapides en premiers, pour réduire le temps total, suivis des plus efficaces pour un meilleur gain total en qualité. L'auteur obtient systématiquement de bons résultats pour maximiser simultanément les critères d'efficacité et de rapidité à l'aide de combinaisons d'algorithmes de déplacement de nœuds.

Temps (s)	Minimum	Moyenne
0.04	0.033	0.401
4.80	0.079	0.483
8.26	0.114	0.503
10.97	0.121	0.509
13.34	0.126	0.512
15.58	0.125	0.514

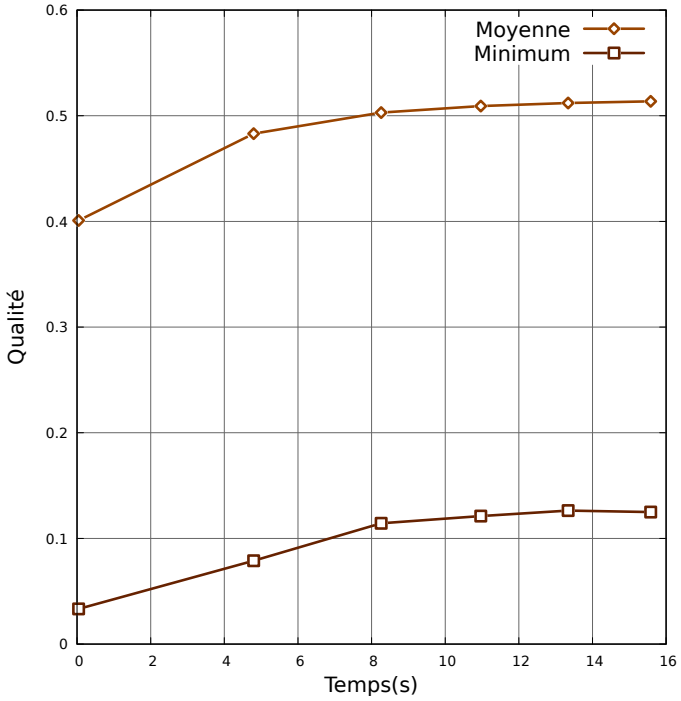


Figure 5.3 Temps et qualités des cinq premières itérations de Nelder-Mead

5.3 Comparaison des techniques d'échantillonnage

Dans la présente section, les différentes techniques d'échantillonnage seront comparées selon leur rapidité et leur précision. Puisque les différentes implémentations d'une même technique donnent des résultats équivalents, on étudiera la précision seulement sur CPU. Par contre, la rapidité des techniques dépend fortement du type d'implémentation. C'est pourquoi on présentera les temps d'échantillonnage séparément pour chaque type d'implémentation.

5.3.1 Précision de la métrique

Le niveau de précision des techniques d'échantillonnage est évalué en adaptant une série de maillages en accentuant de plus en plus l'intensité des chocs simulés par la métrique spécifiée. Puisque les techniques d'échantillonnage utilisées en pratique (recherche locale et texture) n'évaluent pas exactement la métrique spécifiée, on s'attend à ce qu'elles ne permettent pas d'obtenir des maillages de meilleure qualité que ceux obtenus à l'aide de la métrique analytique.

Dix itérations globales du déplacement de nœuds par descente du gradient sont exécutées sur chacun des maillages. En maintenant le facteur de mise à l'échelle à $K = 16$ tout en augmentant progressivement le facteur de compression A, la métrique sinusoïdale présentée à la section 5.2.2 met à l'épreuve la précision des techniques d'échantillonnage. Plus le facteur de compression A est élevé, plus la variation des ratios d'aspects et des densités d'éléments varient fortement à travers le maillage. Ainsi, on observera si les techniques d'échantillonnage réussissent ou non à représenter fidèlement les métriques présentant de fortes variations localisées.

Les maillages initiaux sont générés à partir d'un cube divisé en cinq tétraèdres. Puis une série de modifications topologiques sont appliquées en fonction du facteur de mise à l'échelle et du facteur de compression choisis. Cette étape est effectuée avec la métrique analytique. Une fois que les opérations topologiques ont convergé, la comparaison des techniques d'échantillonnage peut commencer. Toutes les techniques d'échantillonnage sont configurées avec les niveaux de précision par défaut : le maillage de fond de la recherche locale correspond au maillage initial et les textures comportent autant de cellules qu'il y a de nœuds dans les maillages.

Trois exemples de maillages pour différents facteurs de compression sont illustrés à la figure 5.4. Il s'agit de cubes maillés à l'aide de quelques milliers de tétraèdres. Le plus petit maillage correspond au facteur de compression $A = 1$ et comporte 48 444 sommets tandis que le maillage le plus grand correspond au facteur de compression $A = 16$ et comporte 142 469 sommets. Notez que la résolution des supports utilisés par les techniques d'échantillonnage

pour stocker les métriques dépend de la taille des maillages à adapter. Bien que la résolution du maillage de fond ainsi que de la texture augmente plus le facteur de compression augmente, c'est la capacité des techniques à représenter localement la fonction métrique qui est mise à l'épreuve par ce test.

À partir des maillages initiaux, les nœuds sont déplacés par descente du gradient. Le processus est répété pour chaque technique d'échantillonnage. Pour chaque valeur du facteur de compression A, les qualités des pires éléments sont présentées au tableau 5.2 et les moyennes harmoniques des qualités au tableau 5.3.

Les minimums et moyennes des qualités sont donnés avec trois chiffres après la virgule. Puisque les calculs sont effectués en simple précision sur GPU, c'est la précision maximale qui peut être retenue du calcul de la moyenne. Bien qu'il soit possible de donner plus de chiffres significatifs pour les minimums, le minimum des qualités oscille d'une itération à l'autre. Un minimum donné, à une itération donnée, ne nous permet généralement pas de prédire le minimum de l'itération suivante puisque l'on observe souvent des reculs suivis de grandes augmentations. En somme, la croissance du minimum est irrégulière contrairement à la croissance de la moyenne. Toutefois, on s'assurera que les minimums soient similaires entre les techniques d'échantillonnage. Bien que la progression du minimum nous importe peu, on veut s'assurer qu'aucune technique n'ait d'impact négatif significatif sur celle-ci. On se rappellera qu'une poignée d'éléments dégénérés peuvent avoir un impact sur l'ensemble du domaine de la solution.

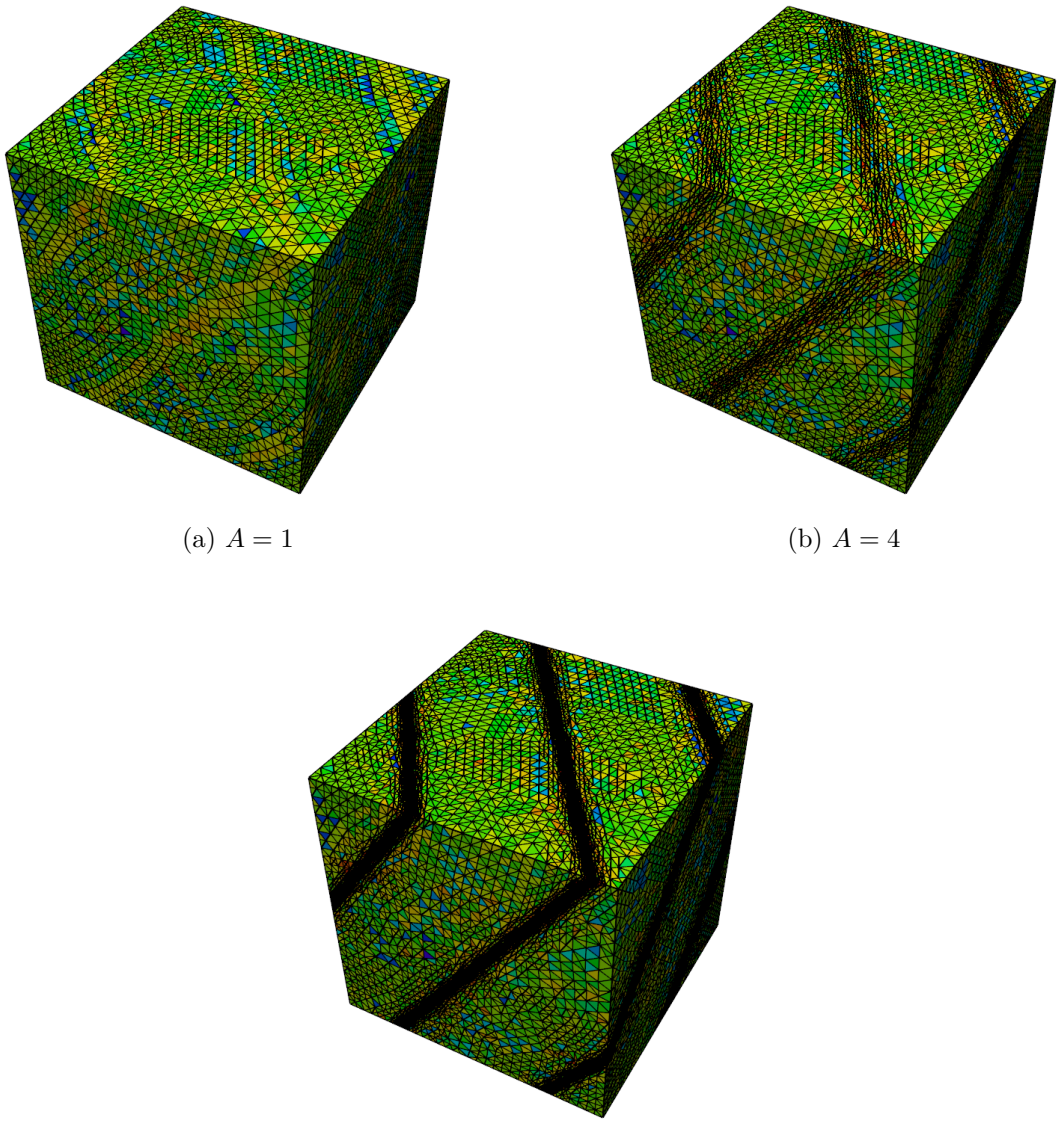
Les valeurs finales pour les pires éléments nous renseignent peu sur la précision réelle des techniques d'échantillonnage. La raison est que l'on tente d'optimiser la moyenne des qualités des voisinages élémentaires, ce qui peut se faire au détriment de quelques éléments dans le voisinage pour le bien de la moyenne. Toutefois, il est important de noter qu'en présence d'une métrique uniforme ($A = 1$), toutes les techniques produisent exactement le même maillage final. Cela s'explique par le fait que toutes les techniques sont capables de représenter une métrique uniforme de manière exacte. Au-delà du facteur de compression uniforme, on observe

Tableau 5.2 Qualités minimales des maillages lissés par descente du gradient en fonction des techniques d'échantillonnage

A	Initial	Analytique	Rech. loc.	Texture
1	0.266	0.308	0.308	0.308
2	0.163	0.294	0.294	0.295
4	0.143	0.231	0.231	0.234
8	0.074	0.175	0.175	0.180
16	0.046	0.118	0.118	0.121

Tableau 5.3 Qualités moyennes des maillages lissés par descente du gradient en fonction des techniques d'échantillonnage

A	Initial	Analytique	Rech. loc.	Texture
1	0.626	0.700	0.700	0.700
2	0.604	0.687	0.687	0.687
4	0.565	0.656	0.656	0.656
8	0.506	0.603	0.603	0.601
16	0.423	0.513	0.512	0.505



(a) $A = 1$

(b) $A = 4$

(c) $A = 16$

Figure 5.4 Maillages obtenus par modifications topologiques dans la métrique sinusoïdale

que la texture semble réussir à tout coup à obtenir de meilleurs gains pour la qualité du pire élément.

Les qualités moyennes nous donnent un meilleur aperçu de la précision des techniques. Jusqu'au facteur de compression $A = 4$, aucun défaut de précision n'est observé de la part du maillage de fond ni de la texture pour les cas de test choisi. Ils obtiennent tous une moyenne de qualité équivalente à celle obtenue à l'aide de la métrique analytique. Toutefois, à partir de $A = 8$, on observe que les imprécisions de la texture commencent à se faire sentir et deviennent même appréciables à $A = 16$.

L'histogramme des qualités des maillages finaux, pour le ratio d'aspect $A = 16$, est donné à la figure 5.5. Les données brutes de l'histogramme sont fournies à l'annexe C. Les distributions des qualités après adaptation sont toujours évaluées dans la métrique analytique. La distribution des qualités finales ayant la forme d'une cloche, on observe bien que toutes les techniques d'échantillonnage réussissent à déplacer la moyenne vers la droite. On voit également que les distributions obtenues par les échantillonnages analytiques et par recherche locale se suivent de très près. L'échantillonnage par texture présente une distribution légèrement différente. Elle est pratiquement bimodale, avec un premier mode à 0.475 et un deuxième à 6.25 qui est quant à lui vis-à-vis celui des deux autres techniques. On en déduit que pour des chocs aussi prononcés, il serait souhaitable d'obtenir un peu plus de précision de la part des textures.

Notez que la résolution des textures peut être ajustée à volonté. Un nombre de cellules équivalent au nombre de nœuds dans le maillage a été choisi comme résolution par défaut, mais l'utilisateur peut diminuer ou, s'il y a encore de l'espace mémoire disponible, augmenter la résolution des textures utilisées. Les textures étant des grilles uniformes, elles sauvent tout l'espace mémoire associé au tableau de nœuds et aux tableaux d'éléments que le maillage de fond doit stocker en plus de la métrique. Il est donc possible d'augmenter quelque peu la résolution des textures sans que l'échantillonneur par texture ne consomme plus de ressources mémoires que l'échantillonneur par recherche locale. Le test qui suit reproduit en partie le test de précision. Ici, on applique le déplacement de nœuds sur un seul maillage tétraédrique en échantillonnant la métrique dans des textures de différentes résolutions. La métrique est configurée avec un facteur de mise à l'échelle $K = 16$ et un facteur de compression $A = 16$. Le nombre de cellules est multiplié par deux à chaque fois. Les qualités finales des maillages adaptés sont données au tableau 5.6.

Le maillage manipulé dans ce test est composé de 142 469 nœuds. Pour ce maillage, la résolution par défaut de la texture est 52^3. La croissance de la qualité moyenne est monotone, ce qui indique qu'une résolution supérieure donne bien un gain en qualité supérieur. Une précision équivalente à celle du maillage de fond est atteinte avec la dernière résolution

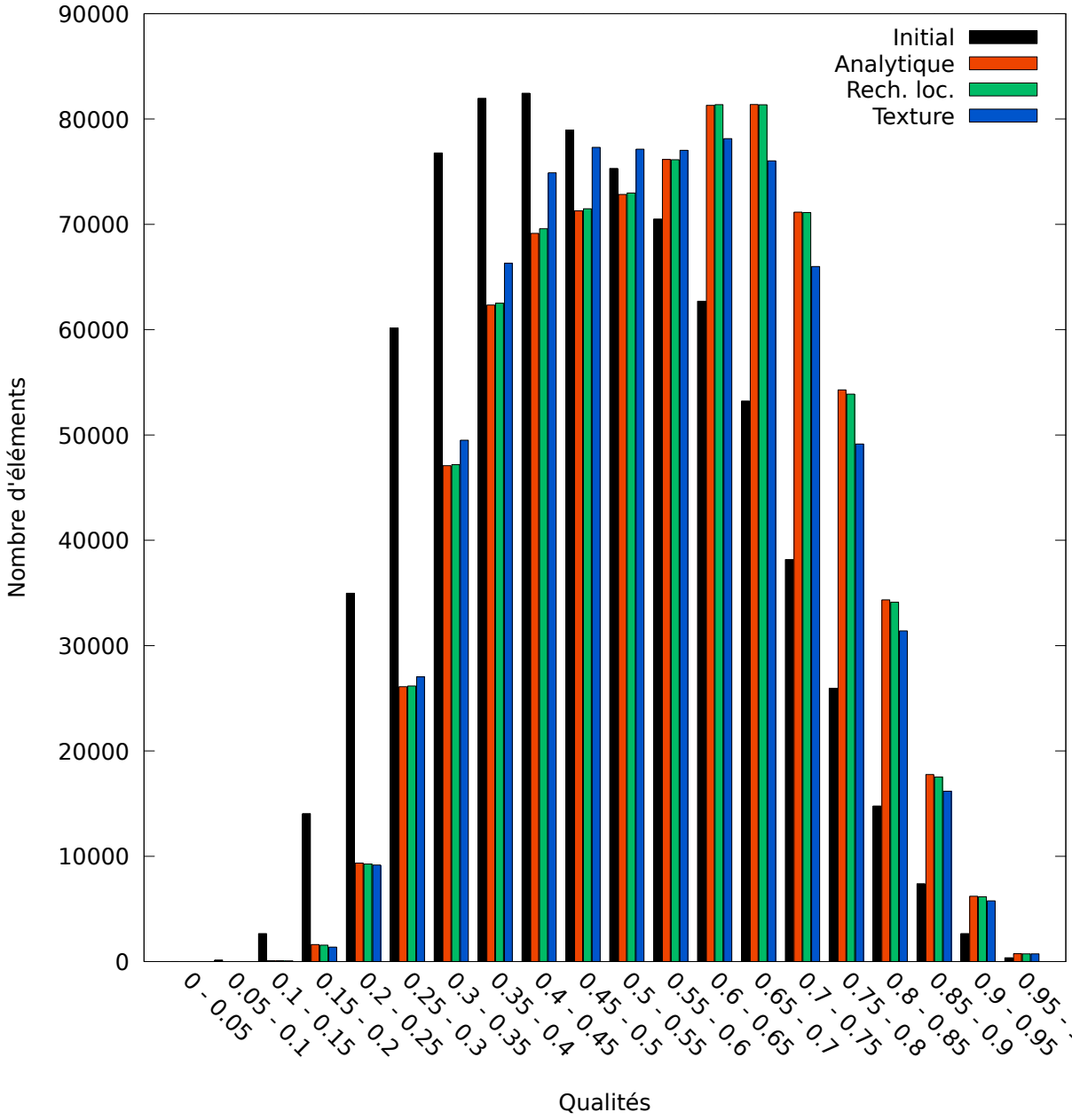


Figure 5.5 Histogramme des qualités des éléments dans les maillages lissés avec un ratio d'aspect $A = 16$

Résolutions	Minimums	Moyennes
Initial	0.046	0.423
16	0.090	0.472
20	0.116	0.478
25	0.123	0.485
32	0.124	0.493
40	0.126	0.500
51	0.123	0.505
64	0.120	0.508
81	0.123	0.510
102	0.120	0.511
128	0.116	0.512

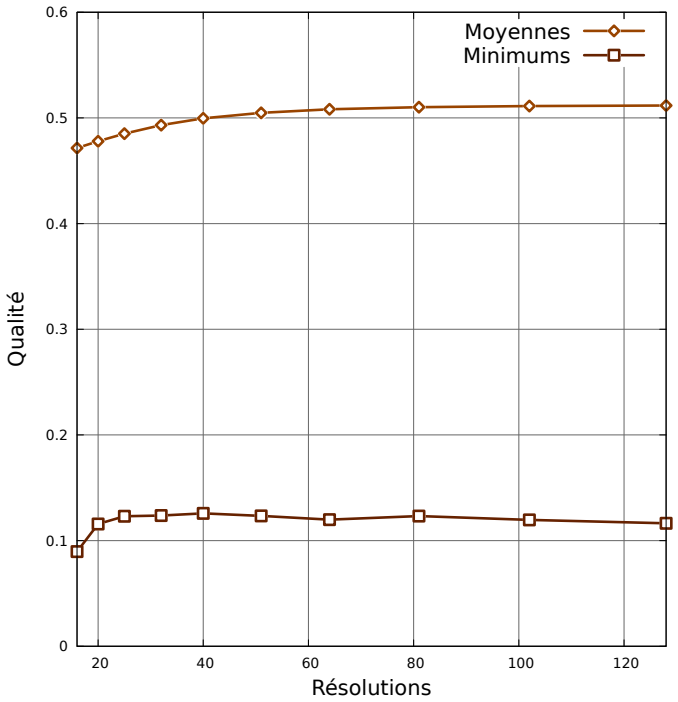


Figure 5.6 Qualités finales du maillage en fonction de la résolution des textures

testée. Cette précision est significativement plus élevée que celle obtenue avec la résolution par défaut, mais consomme également 15 fois plus d'espace mémoire. On préférera utiliser une résolution entre 80 et 100 cellules par côté, car elles offrent de meilleur compromis entre précision et espace mémoire utilisée. Étant donné que les tests qui suivent dans ce chapitre utilisent le facteur de compression $A = 8$, les textures seront configurées avec leur résolution par défaut. À ce niveau de compression, le défaut de précision est à peine perceptible.

5.3.2 Taille des blocs

Avant d'analyser la vitesse des différentes implémentations, il faut déterminer la taille des blocs de calcul qui seront utilisés sur GPU pour évaluer la qualité des maillages. On testera des maillages composés de différents types d'éléments pour visualiser leur impact sur le niveau de parallélisme atteint par le GPU. On utilisera un maillage tétraédrique d'un demi-million de nœuds ainsi qu'un maillage hexaédrique d'un demi-million de nœuds pour lancer les tests. En commençant par un fil par bloc, on augmentera de manière géométrique le nombre de fils d'exécution pour atteindre 256 fils par bloc.

Il ne faut pas oublier que les *Streaming Multiprocessors* manipulent des *warps* de 32 fils. Tout groupe de fils inférieur à 32 est nécessairement enveloppé dans un *warp* complet, les

fils supplémentaires du *warp* étant tout simplement inactifs pendant l'exécution. Mais, les fils inactifs supplémentaires occupent des ressources et limitent le niveau de parallélisme atteignable. Par exemple, si l'on spécifie des blocs d'un seul fil d'exécution, chacun de ces fils sera enveloppé dans un *warp* de 32 fils. Alors, des 2880 cœurs présents sur une carte GeForce GTX 780 Ti, seuls $2880/32 * 1 = 90$ cœurs exécuteront réellement du travail. Pour utiliser les 2880 cœurs simultanément, il faut que la taille des blocs soit nécessairement un multiple de 32.

Une taille de 192 fils a été ajoutée à la liste, bien qu'elle ne soit pas une puissance de 2. 192 correspond au nombre de cœurs que possède chacun des *Streaming Multiprocessors* qu'héberge la GeForce GTX 780 Ti. Il est intéressant de voir si cette taille pourrait faciliter le travail de l'ordonnanceur.

Les temps d'exécution présentés au tableau 5.4 ont été produits par échantillonnage de la métrique dans une texture. Cette technique fait un usage modéré de la mémoire principale et ne présente aucun branchement conditionnel. Elle offre une bonne occasion d'observer le potentiel d'accélération des opérations arithmétiques tout en mettant à l'épreuve la bande passante de la mémoire principale du GPU. Les temps sont donnés à la milliseconde près.

D'après les résultats, il est clair qu'en bas de 16 fils d'exécution, les blocs sont trop petits pour activer tout le parallélisme offert par le GPU. On voit bien la diminution linéaire du temps de calcul en fonction de la taille des blocs. Chaque multiplication de la taille par 2 coupe le temps de calcul par 2. De manière surprenante, 16 semble être une bonne taille du point de vue des implémentations GLSL, bien que les implémentations CUDA soient plus à l'aise avec des tailles de 32 fils. Au-delà de 32 fils, les deux types d'implémentation observent une légère perte d'accélération avant d'atteindre un plateau. Une exception à ce plateau est le bloc de 192 fils en GLSL, qui diminue légèrement les temps d'exécution sans le faire tomber sous les blocs de 16 fils. Donc, faire correspondre la taille des blocs au nombre de cœurs des *Streaming Multiprocessors* ne semble pas être avantageux dans notre cas.

Si le GPU travaille toujours avec des *warps* de 32 fils, pourquoi 16 semble-t-il être une bonne taille de bloc pour l'implémentation GLSL, d'autant plus que l'implémentation CUDA semble toujours favoriser des blocs de 32 fils ? Il est difficile de répondre à cette question sans une connaissance approfondie des pilotes logiciels fournis par le fabricant de la carte graphique. Malheureusement, il est très difficile d'avoir accès à ce type d'information sans compter qu'une telle analyse serait fastidieuse. On devra accepter les résultats comme tels pour faire notre choix de configurations optimales.

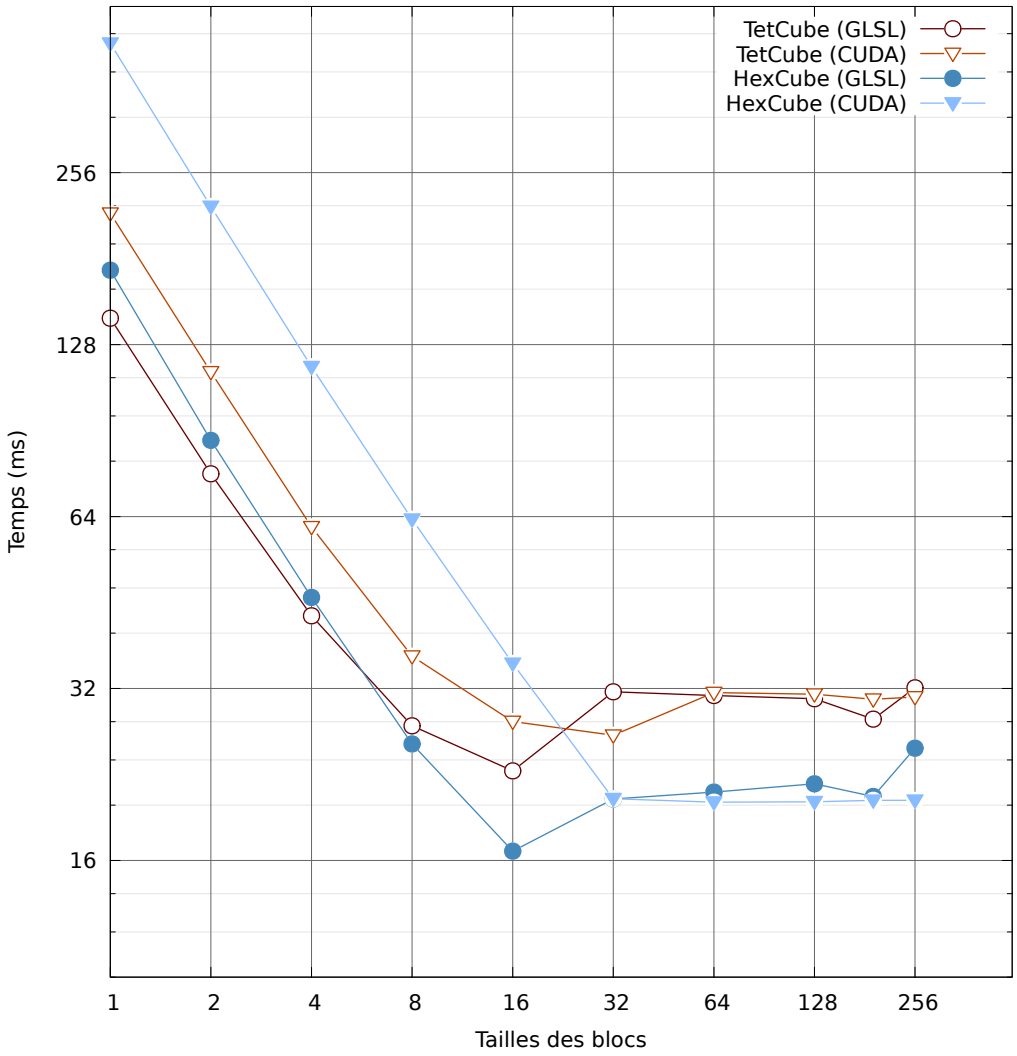


Figure 5.7 Graphique log-log du temps d'évaluation de la qualité en fonction de la taille des blocs de calcul sur GPU

Tableau 5.4 Temps de calcul en millisecondes pour l'évaluation de la qualité en fonction de la taille des blocs de calcul sur GPU

Tailles	TetCube		HexCube	
	GLSL	CUDA	GLSL	CUDA
1	142	217	173	431
2	76	115	87	223
4	43	61	46	117
8	28	36	26	63
16	23	28	17	35
32	32	27	21	21
64	31	31	21	20
128	31	31	22	20
192	28	31	21	20
256	32	31	25	20

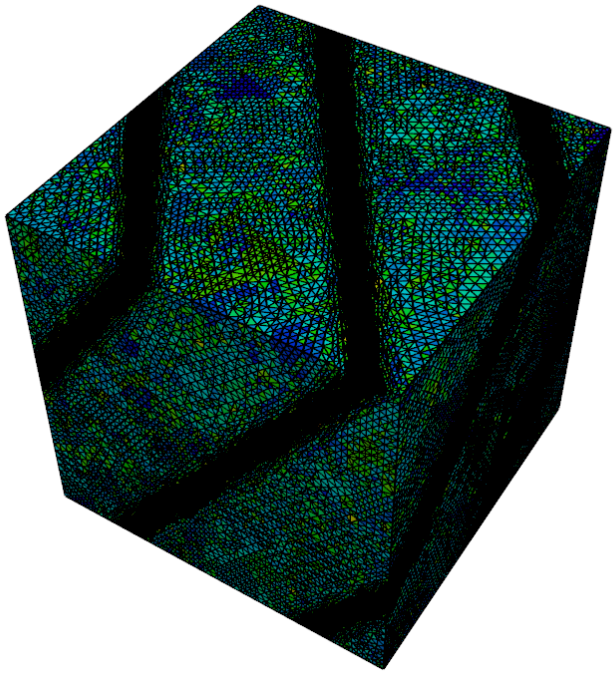
5.3.3 Coût d'échantillonnage

On sait déjà qu'échantillonner la métrique est l'opération la plus coûteuse du processus d'adaptation. Il est donc raisonnable de mesurer la rapidité d'un algorithme de déplacement de nœuds en fonction de la technique d'échantillonnage utilisée. On mesurera ici le temps nécessaire pour calculer la qualité totale d'un maillage, c'est-à-dire la qualité de son pire élément et la moyenne harmonique des qualités de ses éléments. Cela nous permettra d'évaluer quelles sont les techniques d'échantillonnage qui possèdent les plus grands potentiels d'accélération pour les algorithmes de déplacement de nœuds.

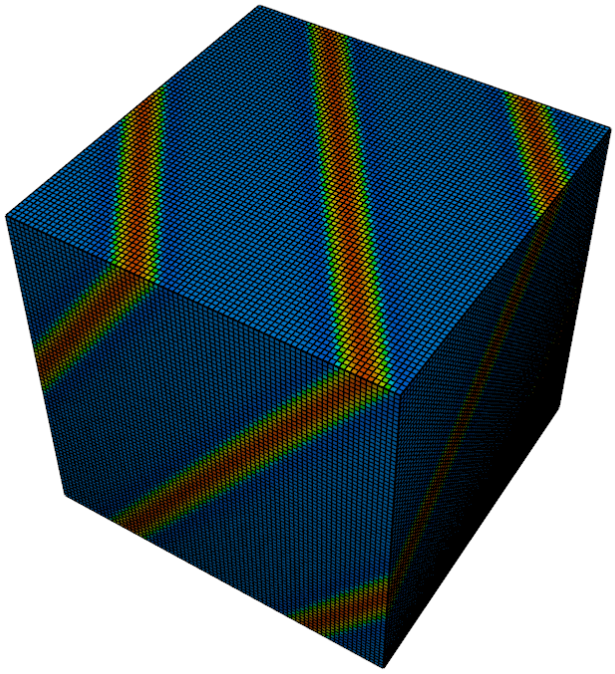
On testera deux maillages synthétiques. Le premier est un maillage tétraédrique en forme de cube d'un demi-million de nœuds (figure 5.8a). Le deuxième est un maillage hexaédrique en forme de cube d'un demi-million de nœuds également (figure 5.8b). La métrique spécifiée est la métrique sinusoïdale de facteurs $K = 28.57$ et $A = 8$. Les temps d'évaluation pour chacune des techniques d'échantillonnage et chacune de leurs implémentations sont donnés aux tableaux 5.5 et 5.6.

Les temps d'évaluation des implémentations séquentielles sont tous du même ordre de grandeur, tous compris entre une à trois secondes. Quant aux implémentations parallèles, puisque tous les éléments des maillages peuvent être évalués simultanément et que la configuration matérielle utilisée possède un processeur à quatre cœurs physiques hébergeant chacun deux cœurs logiques, il n'est pas surprenant d'observer des taux d'accélération environnant les 5x.

Les implémentations GPU présentent des taux d'accélérations beaucoup plus variables. L'implémentation GLSL de l'échantillonnage analytique est celle qui affiche le plus grand taux



(a) Cube de tétraèdres



(b) Cube d'héxaèdres

Figure 5.8 Maillages utilisés pour tester la rapidité des techniques d'échantillonnage

Tableau 5.5 Temps d'évaluation en millisecondes pour un maillage tétraédrique de 500K nœuds

Métriques	Temps (ms)				Accélérations		
	Séquentiel	Parallèle	GLSL	CUDA	Parallèle	GLSL	CUDA
Analytique	2273	404	12	14	5.6	191.1	159.6
Rech. loc.	1188	246	615	108	4.8	1.9	11.0
Texture	1723	343	24	25	5.0	72.9	68.1

Tableau 5.6 Temps d'évaluation en millisecondes pour un maillage hexaédrique de 500K nœuds

Métriques	Temps (ms)				Accélérations		
	Séquentiel	Parallèle	GLSL	CUDA	Parallèle	GLSL	CUDA
Analytique	2934	504	5	21	5.8	551.1	138.0
Rech. loc.	1769	369	inf	312	4.8	0.0	5.7
Texture	1626	344	16	18	4.7	98.8	90.8

d'accélération. L'échantillonnage analytique est la technique qui est à la fois la plus intensive en termes d'opérations arithmétiques, la moins dépendante de la mémoire globale ainsi qu'une de celles qui présentent le moins de branchements conditionnels. Ces trois points expliquent bien les excellentes performances de l'échantillonnage analytique sur GPU.

Notez que l'implémentation GLSL de l'échantillonnage par recherche locale pour le maillage hexaédrique s'est vu attribuer un temps infini puisque l'exécution du noyau n'a pas pu terminer en moins de 5 minutes. Les temps d'exécution de l'échantillonnage par recherche locale ne présentent pas des taux d'accélérations particulièrement grands ni pour le maillage tétraédrique ni pour le maillage hexaédrique, que ce soit en GLSL ou en CUDA. C'est précisément l'instabilité de la technique ainsi que sa faible accélération sur GPU qui a encouragé le développement d'une nouvelle technique d'échantillonnage pour l'adaptation de maillage sur GPU. Il y a deux raisons qui expliquent les mauvaises performances de la recherche locale sur GPU : des lectures fréquentes en mémoire principale ainsi qu'une très grande divergence des fils d'exécution. Comme il a déjà été mentionné, la plupart des fils visiteront un à deux éléments du maillage de fond, tandis que d'autres en visiteront une vingtaine, ralentissant du même coup tout le bloc de calcul auquel ils appartiennent. Cette technique convient certainement mal au modèle d'exécution SIMD.

La nouvelle technique d'échantillonnage proposée est l'échantillonnage par texture. On voit qu'elle n'est pas pire que les autres techniques sur CPU, mais offre des temps d'évaluation plus bas ainsi que des taux d'accélération plus hauts sur GPU que l'échantillonnage par recherche locale. Le succès de cette technique repose sur sa simplicité. Il s'agit uniquement

d'une transformation affine suivie de deux échantillonnages de texture. En d'autres mots : des accès mémoires limités et une absence totale de branchements conditionnels. C'est exactement le type d'exécution que les GPU arrivent à bien paralléliser.

À première vue, les différences de performance entre les implémentations GLSL et CUDA sont difficilement explicables ; d'autant plus que leurs codes sources sont pratiquement identiques, à quelques différences syntaxiques près. Une inspection minutieuse du code assembleur produit pour chacun des langages permettrait de révéler les causes, mais il est très difficile de faire ce type d'analyse en GLSL.

5.3.4 Meilleurs techniques d'échantillonnage

En ayant en mains les résultats de précision et de rapidité des différentes techniques d'échantillonnage présentées dans ce mémoire, on est maintenant prêt à dégager une technique d'échantillonnage optimale par type de processeur. La rapidité est un critère très important pour cette recherche, mais ce critère ne doit pas être satisfait s'il induit une perte substantielle de qualité des maillages adaptés. Pour s'en prémunir, une courte analyse de précision a été conduite sur l'échantillonnage par texture. Bien que Rokos et al. (2011) aient utilisé des textures pour adapter des maillages triangulaires sur GPU, les auteurs ont omis de valider la précision de cette technique, mettant ainsi en doute la pertinence des taux d'accélération qu'ils ont obtenus.

Échantillonnage sur CPU

Parce qu'il s'agit de la technique la plus précise et qu'elle démontre une très grande rapidité pour les maillages tétraédriques, la configuration optimale du CPU échantillonnera la métrique spécifiée par recherche locale dans un maillage de fond. Il ne faut pas oublier qu'il s'agit du format standard pour fournir une métrique spécifiée en adaptation de maillage. Les solutions produites par les solveurs sont stockées aux nœuds du maillage. Pour une solution donnée, plusieurs métriques peuvent être calculées puis intersectées pour produire une métrique spécifiée finale stockée encore une fois aux nœuds du maillage. Le maillage de fond reprend exactement cette représentation, évitant par le fait même toute approximation et diffusion de la métrique spécifiée.

Échantillonnage sur GPU

Les performances relatives des techniques d'échantillonnage diffèrent sur le GPU et c'est pourquoi la texture apparaît comme étant un bien meilleur choix sur ce type de processeur.

Sa rapidité sur CPU ne dépasse pas celle de la recherche locale, mais elle est très bien accélérée par le GPU. De plus, il a été démontré que la texture pouvait atteindre des niveaux comparables de précision à ceux du maillage de fond. Et si pour une configuration donnée, la résolution par défaut de la texture ne suffit pas, il suffit de l'augmenter dans les limites de la mémoire disponible sur la carte graphique.

5.4 Comparaison des algorithmes de lissage

La deuxième composante du pipeline ayant un impact significatif sur l'efficacité et la rapidité du processus d'adaptation est l'algorithme de lissage utilisé. Comme pour les techniques d'échantillonnage, on étudiera les algorithmes de lissage en termes d'efficacité et de rapidité. D'abord, les algorithmes ne convergent pas tous vers la même solution. Prenons par exemple le lissage laplacien qui, sous sa forme classique, produit souvent des tétraèdres inversés. Au contraire, pour des pas de déplacement suffisamment petits et un nombre d'itérations suffisamment grand, les méthodes d'optimisation locale telles que la descente du gradient, Nelder-Mead et l'algorithme de force brute convergeront tous vers la solution optimale. Toutefois, bien que les algorithmes d'optimisation locale convergent éventuellement vers l'optimum, ils le font à des vitesses bien différentes, que ce soit en temps de calcul ou en nombre d'itérations.

Avant de présenter les résultats pour l'efficacité et la rapidité des algorithmes, on exposera deux différences fondamentales entre l'implémentation séquentielle et les implémentations parallèles : l'impact de l'ordre des nœuds déplacés et des différentes représentations des nombres utilisées sur la progression du déplacement.

5.4.1 Écarts des implémentations

Ordre de déplacement de nœuds

Tous les algorithmes de déplacement de nœuds présentés dans ce mémoire travaillent localement sur les voisinages élémentaires d'un maillage. On peut donc les catégoriser comme méthodes de Gauss-Seidel. C'est-à-dire que plusieurs itérations globales sont exécutées sur le maillage entier. Chacune de ces itérations globales est composée d'opérations locales manipulant des voisinages élémentaires. Pour garantir la conformité du maillage final, l'intersection des voisinages élémentaires des nœuds déplacés simultanément doit être vide. C'est exactement pourquoi on utilise le concept d'*ensembles de nœuds indépendants* présenté à la section 2.5.1. De plus, il n'est pas possible de garantir la conformité du maillage sans utiliser la position la plus récente des nœuds voisins lorsque l'on déplace un nœud.

Donc, à l'intérieur d'une itération globale, chaque opération utilise le résultat des opérations précédentes pour produire son propre résultat. Non seulement cette caractéristique des méthodes de Gauss-Seidel accélère la convergence, mais elle représente également une condition nécessaire pour garantir la validité de la solution dans notre cas.

La génération des ensembles de nœuds indépendants change implicitement l'ordre des nœuds déplacés. Les implémentations séquentielles traitent les nœuds dans l'ordre donné par le tableau de nœuds du maillage. Les implémentations parallèles (CPU et GPU) traitent les nœuds en fonction de leur regroupement en ensembles de nœuds indépendants. Tous les nœuds de l'ensemble 1 sont déplacés, suivis des nœuds de l'ensemble 2 et ainsi de suite. Cette différence dans l'ordre des opérations a un impact sur les gains obtenus par chacune des itérations globales, mais a-t-elle un impact sur la vitesse de convergence des algorithmes ?

Cent itérations du déplacement de nœuds par descente du gradient ont été exécutées sur le maillage d'une sphère. Les qualités minimales et moyennes en termes de conformité à la métrique analytique sont comparées entre les implémentations séquentielle et parallèle sur CPU au tableau 5.7. Les résultats des dix premières itérations y sont présentés ainsi que les résultats de la centième itération.

On remarque qu'effectivement il y a une différence dans les qualités des maillages après chaque itération, principalement dans les qualités minimales, mais que celle-ci est négligeable. De plus, il est clair qu'après un grand nombre d'itérations, les deux implémentations donnent toujours des résultats similaires. On observe également à la figure 5.9 que les distributions de qualités dans les maillages finaux sont pratiquement identiques.

Représentation des nombres

L'ordre de déplacement des nœuds est identique pour toutes les implémentations qui déplacent les nœuds parallèlement, que ce soit sur CPU ou GPU. Cependant, leurs résultats ne sont pas identiques. Le tableau 5.8 présente les qualités du même maillage d'une sphère lissé à l'aide de l'implémentation parallèle puis des implémentations GPU du déplacement de nœuds par descente du gradient. L'histogramme des qualités finales est présenté à la figure 5.10. Les données brutes de l'histogramme sont fournies à l'annexe C.

Les différences sont bien inférieures à celles induites par l'ordre de déplacement des nœuds du point de vue des minimums des qualités. Dans ce cas-ci, c'est la précision de la représentation des nombres qui influence les résultats. Puisque le processeur central utilisé fonctionne sur 64 bits, il est tout à fait naturel de représenter tous les nombres à virgule flottante sous forme de *double*s. Par contre, le processeur de la carte graphique utilisée possède bien moins

Tableau 5.7 Conformités minimales et moyennes à la métrique analytique en fonction de l'ordre des nœuds déplacés

Itérations	Minimums		Moyennes	
	Séquentiel	Parallèle	Séquentiel	Parallèle
0	0.001	0.001	0.224	0.224
1	0.038	0.041	0.275	0.275
2	0.036	0.044	0.284	0.284
3	0.043	0.047	0.288	0.288
4	0.048	0.048	0.290	0.290
5	0.050	0.050	0.291	0.291
6	0.050	0.050	0.292	0.292
7	0.050	0.050	0.293	0.293
8	0.050	0.050	0.294	0.294
9	0.051	0.050	0.294	0.294
10	0.051	0.051	0.295	0.295
100	0.064	0.063	0.314	0.314

Tableau 5.8 Conformités minimales et moyennes à la métrique analytique en fonction de l'implémentation parallèle utilisée

Itérations	Minimums			Moyennes		
	Parallèle	GLSL	CUDA	Parallèle	GLSL	CUDA
0	0.001	0.001	0.001	0.224	0.224	0.224
1	0.041	0.041	0.041	0.275	0.275	0.275
2	0.044	0.044	0.044	0.284	0.284	0.284
3	0.047	0.047	0.047	0.288	0.288	0.288
4	0.048	0.048	0.048	0.290	0.290	0.290
5	0.050	0.050	0.050	0.291	0.291	0.291
6	0.050	0.050	0.050	0.292	0.292	0.292
7	0.050	0.050	0.050	0.293	0.293	0.293
8	0.050	0.050	0.050	0.294	0.294	0.294
9	0.050	0.050	0.050	0.294	0.294	0.294
10	0.051	0.051	0.051	0.295	0.295	0.295
100	0.063	0.064	0.063	0.314	0.313	0.313

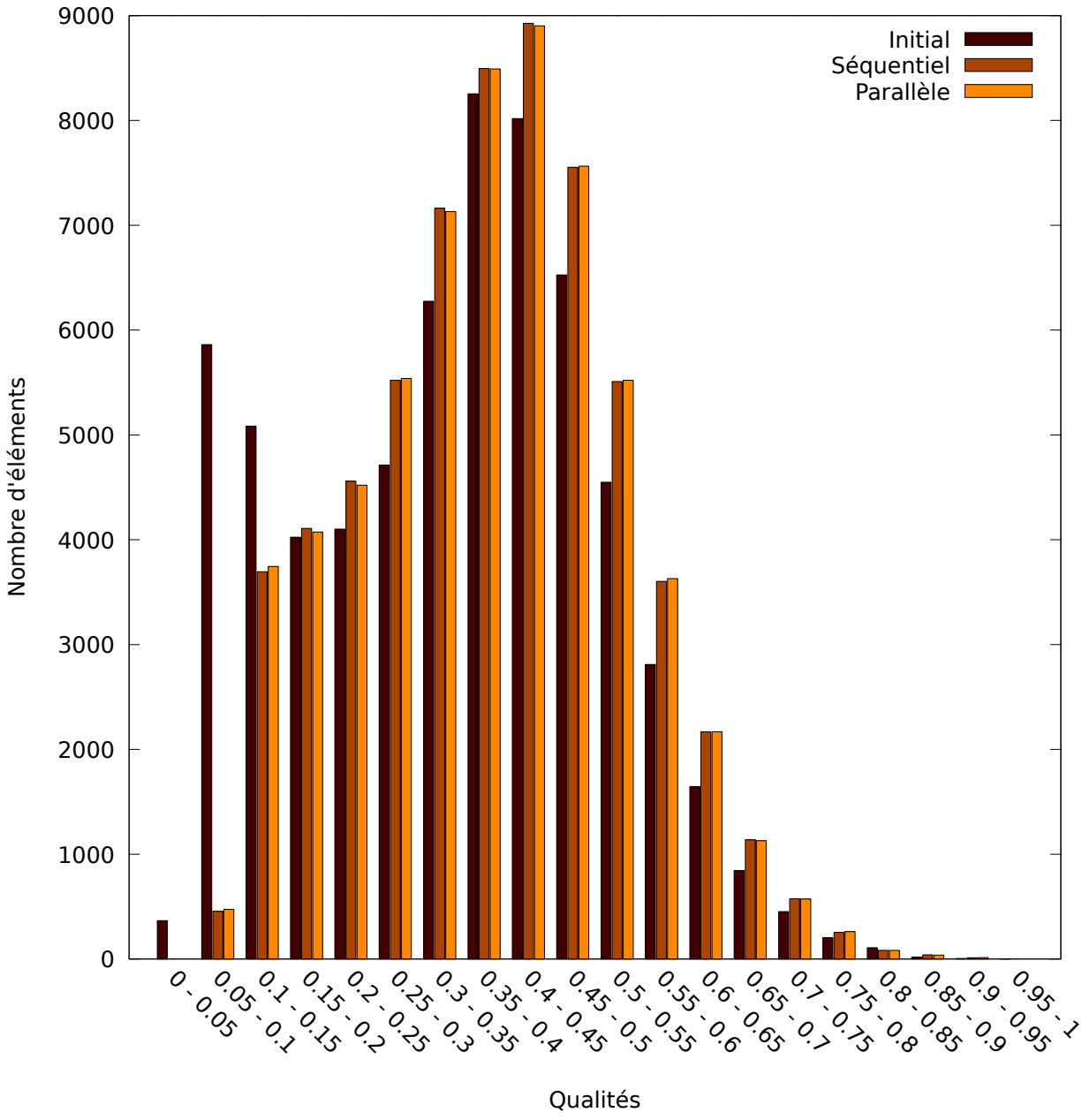


Figure 5.9 Histogrammes des qualités des maillages lissés séquentiellement et parallèlement

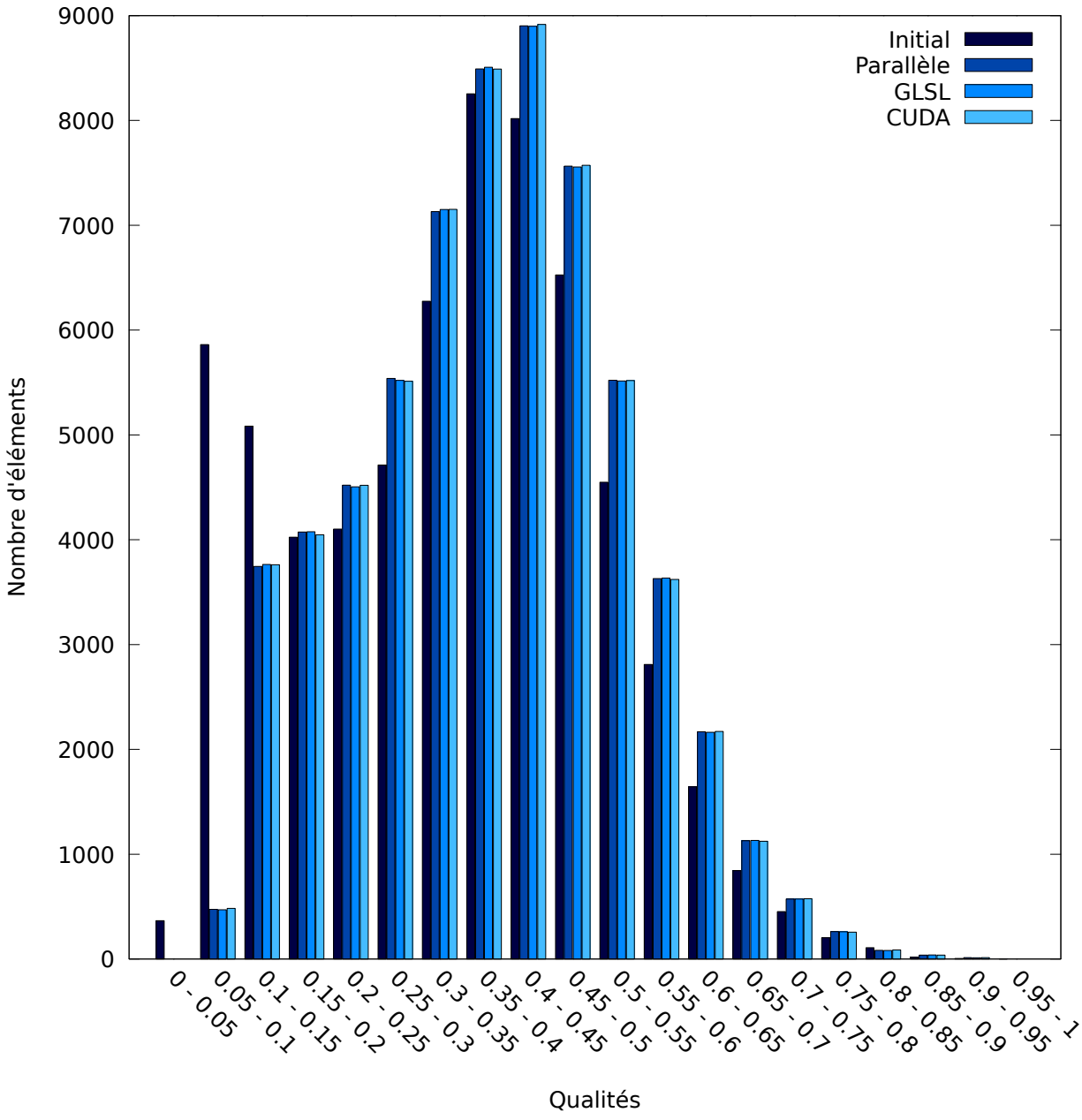


Figure 5.10 Histogrammes des qualités de maillages lissés parallèlement sur CPU et GPU

d'unité de traitement en double précision qu'en simple précision. Même si chaque *Streaming Multiprocessor* possède 3 unités en simple précision pour chaque unité en double précision, la carte graphique utilisée (GTX 780 Ti d'NVIDIA) est physiquement bloquée pour n'utiliser qu'une unité en double précision présente sur 8, réduisant ainsi le ratio de performance à 1/24. D'autres GPU de la même génération permettent d'utiliser toutes les unités en double précision disponibles (ex. : GTX Titan Black et Tesla K40c), mais leur prix est également beaucoup plus élevé. C'est pourquoi seuls les calculs les plus sensibles sont exécutés en double précision dans les présentes versions des implémentations GPU.

D'autres valeurs bénéficieraient d'une représentation en double précision, mais l'impact sur le temps de calcul aurait été trop grand dans le contexte actuel. Par exemple, dans certains cas, il est essentiel de représenter la position des nœuds en double précision, que ce soit à cause du nombre élevé de nœuds dans le maillage ou de la présence de géométries fines sur la frontière du domaine. Pour le bien de cette recherche, il a été jugé préférable d'effectuer la majorité des opérations en simple précision afin de mieux estimer l'accélération que fournirait un vrai accélérateur de calcul GPU tel qu'une Tesla K40c. Toutefois, si l'opportunité se présentait de lancer le logiciel d'adaptation sur un vrai accélérateur de calcul, changer le format des nombres en virgule flottante pour la double précision se ferait sans problème.

5.4.2 Efficacité

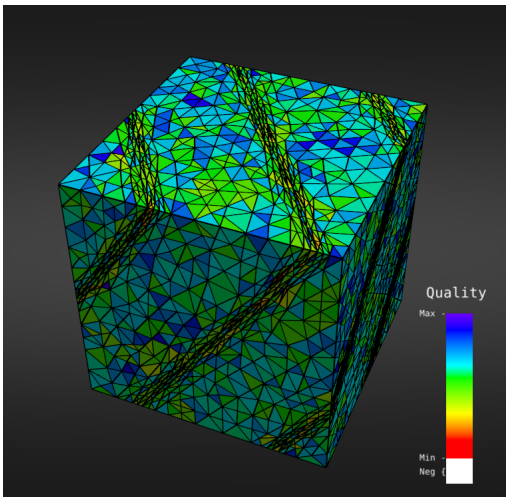
L'efficacité des algorithmes est un facteur déterminant dans le choix des pipelines d'adaptation optimaux. On mesurera l'efficacité des algorithmes selon leur capacité à augmenter la qualité d'un maillage en dix itérations globales. Encore une fois, on se penchera sur la qualité minimale du maillage et la qualité moyenne, bien que la qualité minimale soit peu révélatrice dans la majorité des cas. Les qualités initiale et finale pour un maillage tétraédrique et un maillage hexaédrique en forme de cubes sont données au tableau 5.9. Les maillages initiaux et adaptés peuvent être visualisés à la figure 5.11.

Les résultats sont différents d'un maillage à l'autre. D'un côté, le lissage laplacien à ressorts produit des éléments inversés pour le maillage tétraédrique (d'où la qualité minimale négative et la valeur NaN pour la moyenne harmonique), tandis que ce même lissage laplacien à ressorts réussi a adapté le maillage hexaédrique, mais de manière limitée. La présence d'éléments inversés n'est pas une surprise puisque le lissage laplacien à ressorts ne prend jamais en compte la qualité des voisinages élémentaires pour déplacer les nœuds.

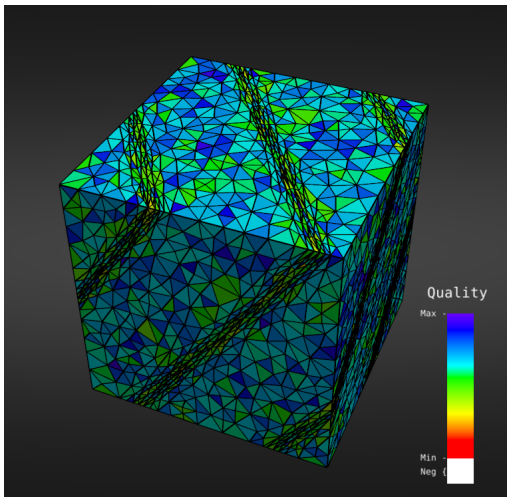
Bien que le lissage laplacien de qualité suive la même tendance que la version à ressorts, il donne de meilleurs résultats. D'abord, le lissage laplacien de qualité ne produit aucun élément inversé dans le maillage tétraédrique, sans toutefois augmenter de beaucoup la qualité du pire

Tableau 5.9 Qualités minimale et moyenne finales pour chaque algorithme de déplacement de nœuds

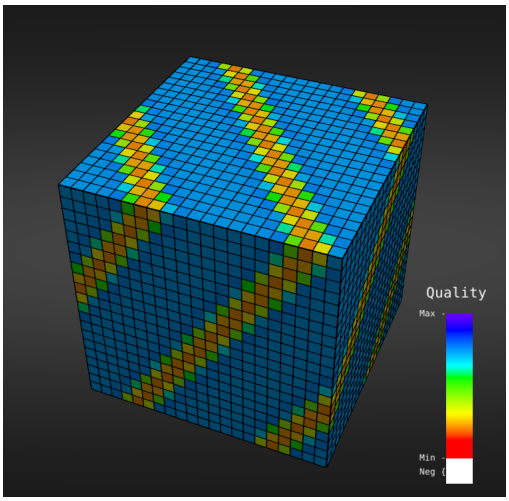
Métriques	TetCube		HexCube	
	Minimums	Moyennes	Minimums	Moyennes
Initial	0.168	0.541	0.214	0.541
Laplace à ressorts	-0.141	NaN	0.309	0.649
Laplace de qualité	0.182	0.600	0.254	0.663
Force brute	0.242	0.637	0.260	0.691
Nelder-Mead	0.237	0.637	0.457	0.694
Descente du gradient	0.239	0.632	0.455	0.698



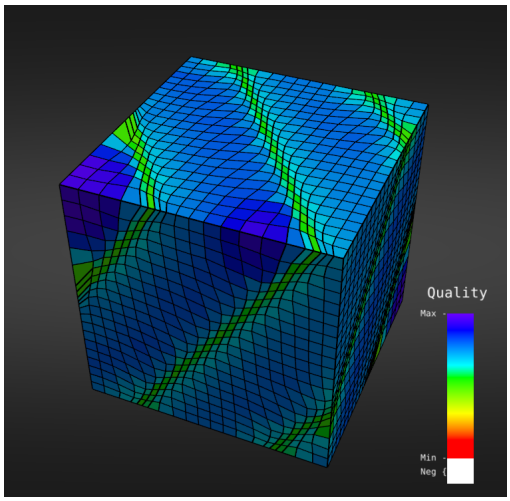
(a) Tétraédrique intial



(b) Tétraédrique adapté



(c) Hexaédrique intial



(d) Hexaédrique adapté

Figure 5.11 Maillages initiaux et adaptés à la métrique sinusoïdale par Nelder-Mead

élément. Il réussit également à augmenter la moyenne des qualités, mais d'une manière bien modeste par rapport aux algorithmes d'optimisation locale. Il suit la même tendance pour le maillage hexaédrique.

L'algorithme de force brute est celui qui est le plus sensible au pas de déplacement. Par expérience, il a été observé que les petits pas ont tendance à favoriser l'adaptation de maillages tétraédriques tandis que les grands pas favorisent les maillages hexaédriques. Le pas de déplacement choisi est un compromis entre les deux types de maillage. Son efficacité est comparable aux deux autres algorithmes d'optimisation locale pour le maillage tétraédrique et le maillage hexaédrique, quoique la qualité minimale soit un peu basse pour ce dernier.

Quant à la descente du gradient et Nelder-Mead, on peut dire qu'ils sont équivalents. Pour les cas de test proposés, l'un est plus efficace pour le maillage tétraédrique tandis que l'autre l'est plus pour le maillage hexaédrique. Ce sont les deux algorithmes les plus efficaces de tous ceux étudiés. De plus, ils se sont montrés aussi efficaces, peu importe le type d'éléments à adapter.

Finalement, on peut conclure qu'il est préférable de choisir un algorithme de déplacement de nœuds par optimisation locale plutôt qu'un algorithme basé sur le lissage laplacien. Vu l'équivalence des méthodes d'optimisation locale, on doit attendre l'analyse des vitesses pour choisir l'algorithme du pipeline optimal.

5.4.3 Taille des blocs

Comme pour l'échantillonnage de la métrique, la taille des blocs de calcul sur le GPU a un effet déterminant sur la rapidité des algorithmes de déplacement de nœuds. Encore une fois, on analysera l'effet de la taille des blocs en fonction du type d'éléments qui composent le maillage. On remarquera que, contrairement aux implémentations de l'évaluation de la qualité, les blocs de calcul prendront différentes formes, en fonction de l'espace de parallélisation. La majorité partage une forme unidimensionnelle où plusieurs nœuds sont déplacés simultanément (un nœud par fil, section 4.3.1). Sinon, on retrouve des dispositions rectangulaires pour les algorithmes de descente du gradient multiélément, multiposition et multiaxe (un nœud par bloc, section 4.3.2). Le présent test s'attarde uniquement aux blocs unidimensionnels. Les tailles des blocs rectangulaires seront ajustées manuellement.

Les graphiques 5.12 et 5.13 présentent les temps d'exécution pour la descente du gradient et Nelder-Mead. Le premier graphique correspond aux progressions des vitesses pour un maillage tétraédrique de 175K nœuds tandis que le deuxième correspond au maillage hexaédrique de 175K nœuds.

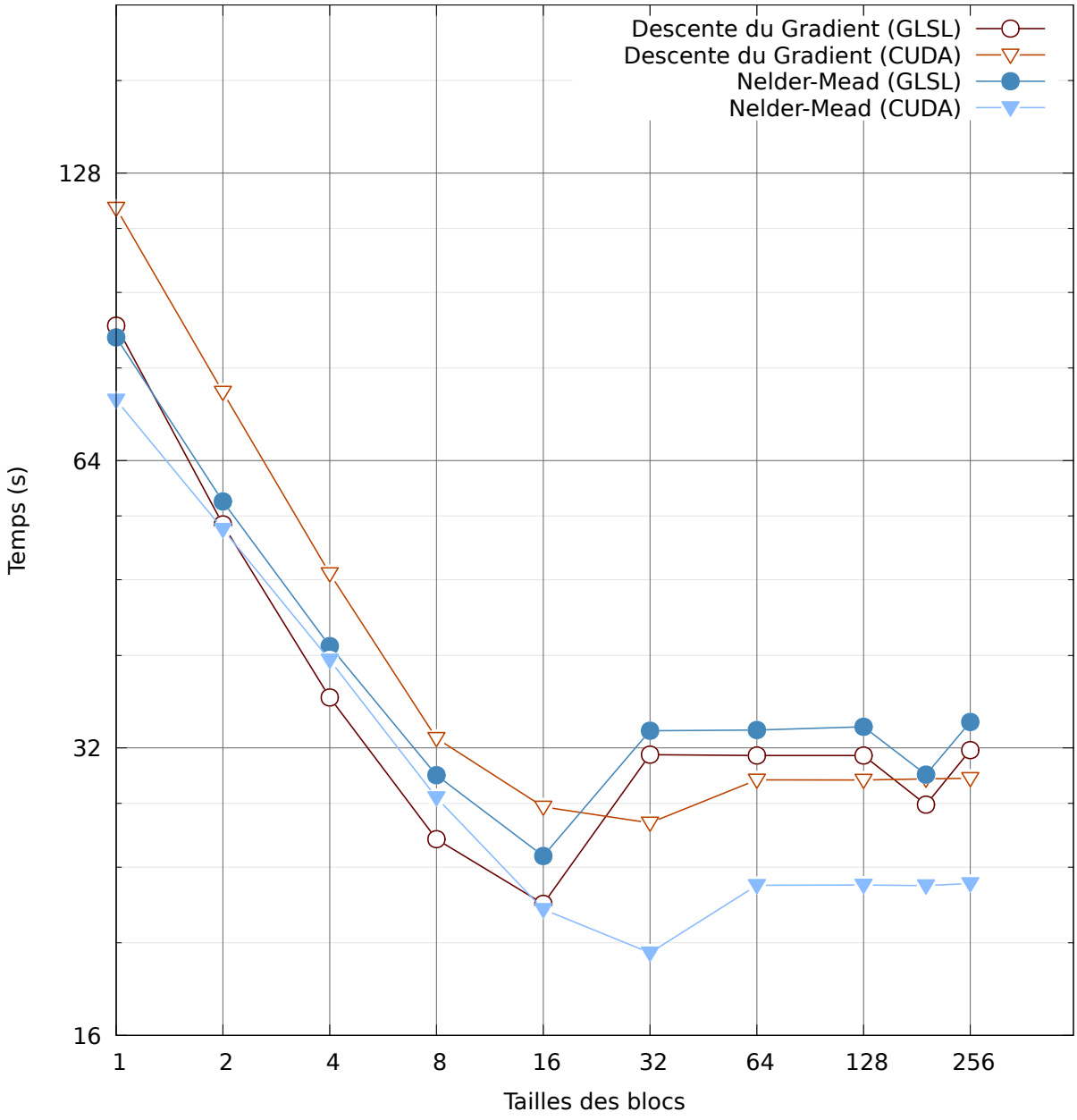


Figure 5.12 Graphique log-log du temps de calcul du lissage d'un maillage tétraédrique en fonction de la taille des blocs de calcul

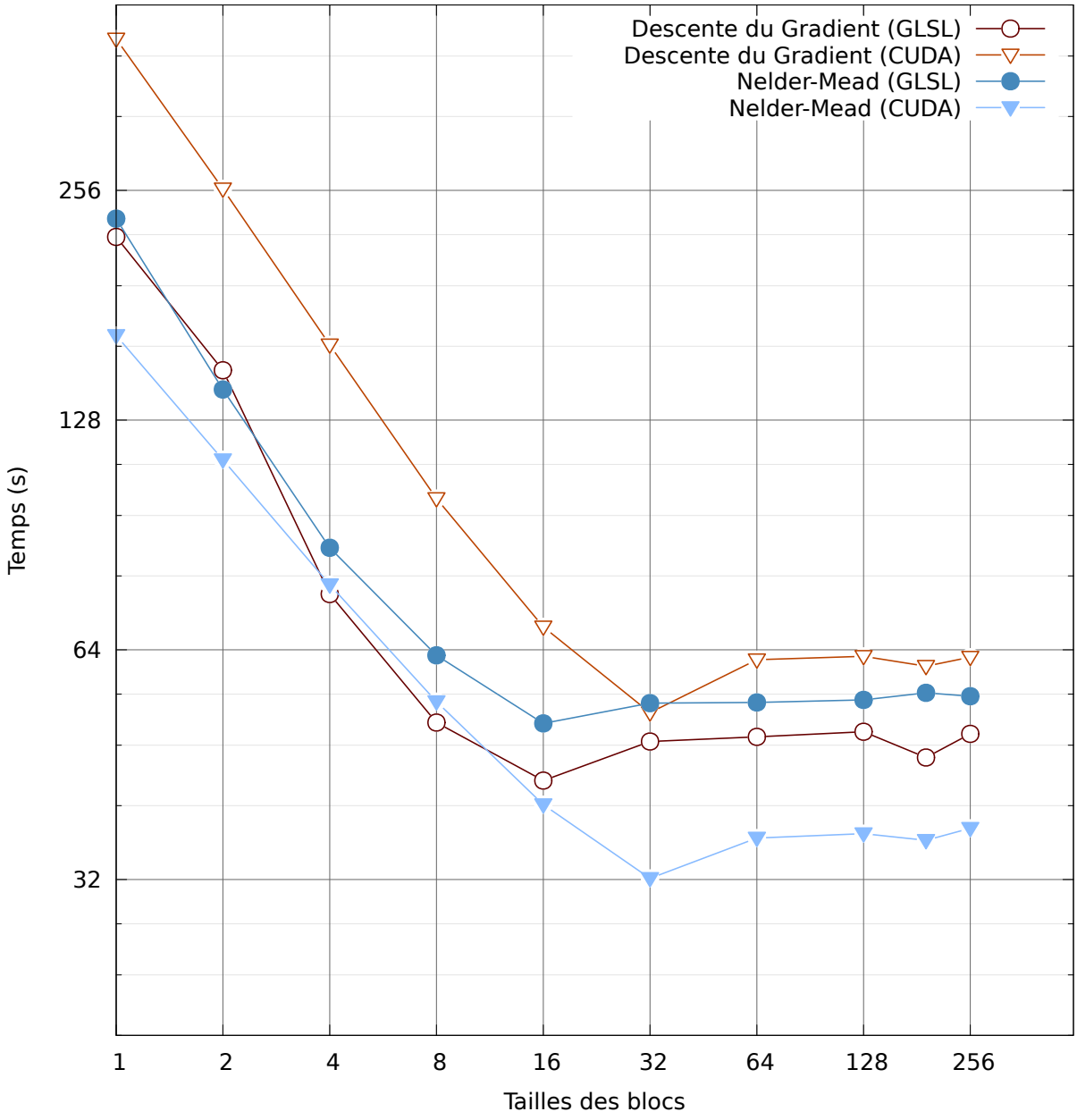


Figure 5.13 Graphique log-log du temps de calcul du lissage d'un maillage hexaédrique en fonction de la taille des blocs de calcul

On observe essentiellement la même tendance que pour l'évaluation de la qualité. C'est-à-dire que les implémentations GLSL favorisent des tailles de blocs de 16 nœuds tandis que les implémentations CUDA préfèrent des tailles de 32 nœuds, autant pour le maillage tétraédrique que le maillage hexaédrique, et ce autant pour la descente du gradient que pour Nelder-Mead.

Puisque les implémentations GPU de la descente du gradient utilisant plusieurs espaces de parallélisation possèdent des tailles de blocs particulières, elles ont été optimisées manuellement. Les tailles de blocs choisis sont présentées aux tableaux 5.10, 5.11 et 5.12. Ces tailles représentent un certain compromis entre les besoins des maillages tétraédriques et les maillages hexaédriques. Il a été remarqué que les algorithmes préféraient traiter des petits nombres de nœuds par blocs pour les maillages tétraédriques alors qu'ils acceptaient un grand nombre de nœuds avant de perdre de la vitesse pour les maillages hexaédriques. On peut expliquer cette différence par le fait qu'il y a une plus grande uniformité entre les voisinages élémentaires formés d'hexaèdres que de tétraèdres, induisant ainsi une plus faible divergence dans l'exécution des noyaux sur GPU.

Avec l'information dont on dispose sur les implémentations GPU des différentes versions de la descente du gradient, il n'est pas possible d'identifier les raisons pour lesquelles on observe de si grandes différences entre les tailles de blocs idéales en GLSL et CUDA. D'abord, les implémentations CUDA sont toujours plus rapides avec un petit nombre de fils pour traiter les nœuds en parallèle (4 contre 32). De plus, pour la version multiaxe, l'implémentation CUDA est plus rapide avec un petit nombre de fils pour traiter les éléments en parallèle que la version GLSL (8 contre 32). On a observé également que ces comportements sont valides autant pour les maillages tétraédriques que les maillages hexaédriques.

Lors des tests de rapidité de la section suivante, on utilisera les tailles de blocs idéales présentées ici. D'une part, pour les implémentations parallélisant l'exécution seulement en fonction des nœuds, des blocs de 16 fils seront utilisés pour les implémentations GLSL tandis que des blocs de 32 fils seront utilisés pour les implémentations CUDA. Pour les versions multiélément, multiposition et multiaxe, on utilisera les tailles de blocs données aux tableaux 5.10, 5.11 et 5.12 directement.

Tableau 5.10 Descente du gradient multiélément : dimensions des blocs de calcul

Nœuds		Éléments	
GLSL	CUDA	GLSL	CUDA
32	4	8	8

Tableau 5.11 Descente du gradient multiposition : dimensions des blocs de calcul

Nœuds		Positions	
GLSL	CUDA	GLSL	CUDA
32	4	8	8

Tableau 5.12 Descente du gradient multiaxe : dimensions des blocs de calcul

Positions		Éléments	
GLSL	CUDA	GLSL	CUDA
8	8	32	8

5.4.4 Rapidité

Maintenant que l'on connaît l'efficacité relative des algorithmes de lissage à l'étude et que l'on a établi une base de comparaison équitable entre les implémentations en déterminant les tailles de blocs idéales pour les noyaux de calcul, on peut maintenant s'attarder à la rapidité des algorithmes. Encore une fois, on mesurera les temps de calcul sur un maillage tétraédrique puis un maillage hexaédrique, tous deux de 175K nœuds, toujours en utilisant la métrique sinusoïdale présentée en début de chapitre.

Les résultats présentés aux tableaux 5.13 et 5.14 correspondent aux temps nécessaires pour exécuter dix itérations globales de chaque algorithme. Les versions supplémentaires de la descente du gradient ont été abrégées par DG (Descente du Gradient) pour économiser de l'espace. Ces temps incluent l'évaluation initiale de la qualité du maillage, ainsi que l'évaluation finale en fonction de la métrique analytique. De plus, dans le cas des implémentations GPU, ces temps incluent également la copie du maillage et de la métrique (c.-à-d. les textures) du CPU vers le GPU en début de processus ainsi que la mise à jour des positions des nœuds sur CPU en fin de processus. Donc, peu importe l'implémentation utilisée, on se retrouve toujours avec le maillage adapté en mémoire principale prêt à être utilisé par le CPU.

Avant de s'intéresser à chaque algorithme individuellement, il est important de noter que toutes les implémentations parallèles sont approximativement 5 fois plus rapides que les implémentations séquentielles. Étant donné que la configuration matérielle utilisée comporte quatre cœurs physiques comportant chacun deux cœurs logiques, cela signifie que les coûts de gestion des fils d'exécution sont négligeables. Quant aux accélérations fournies par les implémentations GPU, elles sont toujours supérieures à celle des implémentations parallèles et sont de l'ordre de 10x à 100x. Il est donc clair qu'il est possible d'accélérer l'exécution du déplacement de nœuds sur GPU au-delà de ce qu'il est possible de faire sur CPU.

Tableau 5.13 Temps de calcul et taux d'accélération des algorithmes de déplacement de nœuds pour 10 itérations globales sur un maillage tétraédrique de 175K de nœuds

Algorithmes	Temps (s)				Accélérations		
	Séquentiel	Parallèle	GLSL	CUDA	Parallèle	GLSL	CUDA
Laplace à ressorts	38.96	7.03	0.62	0.53	5.5	62.7	73.7
Laplace de qualité	109.36	21.60	5.01	5.03	5.1	21.8	21.7
Force brute	1297.38	273.53	45.75	35.20	4.7	28.4	36.9
Nelder-Mead	255.36	54.27	24.80	19.66	4.7	10.3	13.0
Descente du gradient	467.12	99.56	22.06	27.15	4.7	21.2	17.2
DG multiélément	"	"	16.34	10.53	"	28.6	44.4
DG multiposition	"	"	12.17	10.52	"	38.4	44.4
DG multiaxe	"	"	13.62	9.81	"	34.3	47.6

Tableau 5.14 Temps de calcul et taux d'accélération des algorithmes de déplacement de nœuds pour 10 itérations globales sur un maillage hexaédrique de 175K de nœuds

Algorithmes	Temps (s)				Accélérations		
	Séquentiel	Parallèle	GLSL	CUDA	Parallèle	GLSL	CUDA
Laplace à ressorts	28.65	5.53	0.44	0.35	5.2	65.3	82.7
Laplace de qualité	503.42	105.21	8.54	8.96	4.8	58.9	56.2
Force brute	7338.81	1572.70	124.04	118.43	4.7	59.2	62.0
Nelder-Mead	938.81	199.51	51.20	32.30	4.7	18.3	29.1
Descente du gradient	2765.64	576.55	43.13	52.91	4.8	64.1	52.3
DG multiélément	"	"	30.54	26.38	"	90.6	104.8
DG multiposition	"	"	28.99	26.64	"	95.4	103.8
DG multiaxe	"	"	51.01	25.19	"	54.2	109.8

Sans surprise, les algorithmes de déplacement de nœuds basés sur le lissage laplacien sont extrêmement rapides. Déjà, l'implémentation séquentielle de la version à ressorts s'exécute en moins d'une minute, les implémentations GPU sont complétées en moins d'une seconde. La version de qualité est environ trois fois plus lente pour le maillage tétraédrique et près de vingt fois plus lente pour le maillage hexaédrique, mais reste toutefois plus rapide que tous les algorithmes d'optimisation locale. Malheureusement, puisque le lissage laplacien est très peu efficace pour augmenter la qualité des maillages tridimensionnels, sa très grande rapidité ne nous amène pas à le reconsidérer dans le choix des pipelines d'adaptation optimaux.

L'algorithme de force brute offre de meilleures accélérations sur GPU que les versions standards de la descente du gradient et de Nelder-Mead. Malheureusement, parce que son implémentation séquentielle est très lente, ses implémentations GPU n'arrivent pas tout à fait à rejoindre la rapidité des autres algorithmes d'optimisation locale. Bref, bien que ses performances soient appréciables, l'algorithme de force brute n'arrive à se démarquer ni par son efficacité ni par sa vitesse.

La descente du gradient et Nelder-Mead sont un peu plus difficiles à départager. La descente du gradient est plus lente que Nelder-Mead du point de vue séquentiel, mais les deux ont des rapidités comparables sur GPU. Malgré tout, pour les deux types de maillages, l'implémentation CUDA de Nelder-Mead est gagnante. Toutefois, lorsque l'on considère les versions multiélément, multiposition et multiaxe de la descente du gradient, la descente du gradient dépasse Nelder-Mead et culmine avec l'implémentation CUDA de la version multiaxe.

En analysant les rapidités des implémentations CUDA des versions multiélément, multiposition et multiaxe de la descente du gradient, on remarque que les versions multiélément et multiposition sont pratiquement aussi rapides, juste derrières la version multiaxe. Dans le cas du maillage tétraédrique, la version multiaxe atteint une accélération de 47.6x et est deux fois plus rapide que Nelder-Mead. Quant au maillage hexaédrique, cette version atteint une accélération de 109.8x, ce qui représente le plus haut taux d'accélération observé.

En analysant ces résultats, la question s'est posée de savoir s'il était possible de paralléliser Nelder-Mead selon d'autres espaces de parallélisation. Contrairement à la descente du gradient, une seule position d'évaluation est connue à la fois par l'algorithme, ce qui élimine l'espace des positions. Par contre, l'évaluation de la qualité d'un voisinage élémentaire est identique dans tous les algorithmes. S'il a été possible d'utiliser l'espace des éléments pour la descente du gradient, il n'est donc pas impossible d'utiliser cet espace pour Nelder-Mead.

5.4.5 Nelder-Mead multiélément

Développer une version multiélément pour Nelder-Mead revient à appliquer la transformation obtenue par la version multiélément de la descente du gradient. C'est-à-dire que la boucle qui évalue tour à tour la qualité des éléments d'un voisinage élémentaire doit être divisée en deux étapes. On attribue un sous-ensemble du voisinage élémentaire du nœud déplacé à chacun des fils. Chaque fil doit donc évaluer la qualité de son sous-ensemble d'éléments, déterminer la qualité de son pire élément et calculer la moyenne harmonique partielle de ses éléments. Puis, les pires qualités sont assignées à une variable partagée atomique qui ne retiendra que la pire qualité du voisinage complet et les moyennes harmoniques partielles sont accumulées dans une autre variable partagée atomique. Telle que définie à la section 2.4.2, la qualité finale du voisinage élémentaire est donnée par la pire qualité si celle-ci est négative ou la moyenne harmonique des qualités sinon.

Similairement à la version multiélément de la descente du gradient, il est nécessaire de déterminer les tailles optimales des blocs de calcul pour offrir une base de comparaison juste. Ces tailles, présentées au tableau 5.15, ont été ajustées manuellement en testant plusieurs types de maillages. Pour une fois, les implémentations GLSL et CUDA s'accordent sur la taille des blocs à utiliser.

Une fois les tailles de blocs optimales déterminées, les algorithmes ont été lancés sur les maillages tétraédrique et hexaédrique utilisés précédemment pour évaluer la performance de cette nouvelle version. Les tableaux 5.16 et 5.17 présentent les temps d'exécution de la version NM (Nelder-Mead) multiélément et les compare à la version standard de Nelder-Mead ainsi qu'à la version standard et multiaxe de la descente du gradient. Mis à part les temps de la nouvelle version, tous les temps sont tirés du test de rapidité précédent. Bref, ce que l'on tente de déterminer ici est si la version multiélément de Nelder-Mead est plus rapide que la version multiaxe de la descente du gradient.

Non seulement la version multiélément de Nelder-Mead est près de trois fois plus rapide que sa version originale, mais elle est également plus rapide que la meilleure version de la descente du gradient. L'implémentation CUDA de la version multiélément de Nelder-Mead est sans contredit la plus rapide de toutes les implémentations des algorithmes d'optimisation locale.

Tableau 5.15 Nelder-Mead multiélément : dimensions des blocs de calcul

Nœuds		Éléments	
GLSL	CUDA	GLSL	CUDA
4	4	8	8

Tableau 5.16 Comparaison de la descente du gradient et Nelder-Mead sur un maillage tétraédrique

Algorithmes	Temps (s)				Accélérations		
	Séquentiel	Parallèle	GLSL	CUDA	Parallèle	GLSL	CUDA
Nelder-Mead	255.36	54.27	24.80	19.66	4.7	10.3	13.0
NM multiélément	"	"	14.84	6.98	"	17.2	36.6
Descente du gradient	467.12	99.56	22.06	27.15	4.7	21.2	17.2
DG multiaxe	"	"	13.62	9.81	"	34.3	47.6

Tableau 5.17 Comparaison de la descente du gradient et Nelder-Mead sur un maillage hexaédrique

Algorithmes	Temps (s)				Accélérations		
	Séquentiel	Parallèle	GLSL	CUDA	Parallèle	GLSL	CUDA
Nelder-Mead	938.81	199.51	51.20	32.30	4.7	18.3	29.1
NM multiélément	"	"	24.84	12.20	"	37.8	77.0
Descente du gradient	2765.64	576.55	43.13	52.91	4.8	64.1	52.3
DG multiaxe	"	"	51.01	25.19	"	54.2	109.8

5.4.6 Meilleurs techniques de lissage

Le choix de la technique de lissage optimale fut moins évident à faire que le choix de la technique d'échantillonnage. D'abord parce que tous les algorithmes d'optimisation locale ont des efficacités plus ou moins équivalentes, tout en présentant une certaine variabilité selon le type de maillage adapté. Mais aussi parce que tant de versions sur GPU ont été développées pour la descente du gradient et Nelder-Mead. Toutefois, on peut dire qu'un des algorithmes s'est suffisamment démarqué, autant sur le CPU que le GPU.

De prime abord, les algorithmes basés sur le lissage laplacien ont été écartés puisque la version à ressort a tendance à produire des éléments inversés, et donc à détruire la qualité du maillage. De son côté, le lissage laplacien de qualité n'offre pas d'aussi bons gains que les algorithmes d'optimisation locale. Malgré leur très grande rapidité, le Laplace à ressorts et le Laplace de qualité ne feront pas partie des pipelines optimaux.

Bien que l'algorithme de force brute donne des gains de qualité équivalents à la descente du gradient et Nelder-Mead, autant pour les maillages tétraédriques qu'hexaédriques, cet algorithme est tellement lent sur le CPU qu'il ne peut être retenu pour ce pipeline. Quant au GPU, il présente de très bonnes accélérations, mais son implémentation sérielle est tellement lente que ses implémentations GLSL et CUDA arrivent à peine à rattraper les autres algorithmes. On écartera donc l'algorithme de force brute du pipeline GPU optimal également.

Cela nous laisse la descente du gradient et Nelder-Mead. Les deux algorithmes ont leurs avantages en terme d'efficacité. La descente du gradient nous a démontré plus d'aptitude à adapter le maillage hexaédrique tandis que Nelder-Mead a obtenu le plus grand gain pour le maillage hexaédrique. Le choix entre la descente du gradient et Nelder-Mead sur CPU et GPU se fera sur la base de leur rapidité relative.

Lissage sur CPU

Nelder-Mead est le plus rapide sur CPU. Il s'est montré deux fois plus rapide sur le maillage tétraédrique ainsi que trois fois plus rapide sur le maillage hexaédrique que la descente du gradient. Il sera jumelé à l'échantillonnage par recherche locale pour former le pipeline optimal d'adaptation sur CPU.

Lissage sur GPU

Plusieurs versions de la descente du gradient et de Nelder-Mead ont été développées pour le GPU en tirant profit des espaces de parallélisation disponibles pour chacun d'eux. La descente du gradient s'est déclinée en quatre versions selon les espaces suivants : standard $\{Noeuds\}$, multiélément $\{Noeuds \times Éléments\}$, multiposition $\{Noeuds \times Position\}$ et multiaxe $\{Noeuds \times Éléments \times Positions\}$. Puisqu'il n'est pas possible de créer une version multiposition de Nelder-Mead, cet algorithme s'est décliné en seulement deux versions : standard $\{Noeuds\}$ et multiélément $\{Noeuds \times Éléments\}$.

Avant la découverte de la version multiélément de Nelder-Mead, les versions multiélément, multiposition et multiaxe de la descente du gradient obtenaient les plus grandes accélérations tandis que Nelder-Mead obtenait les plus faibles accélérations de tous les algorithmes d'optimisation locale. Cette différence d'accélération est directement liée à la divergence des fils d'exécution. Nelder-Mead comporte plusieurs branchements conditionnels ainsi qu'une boucle d'itération locale de longueur variable. Au contraire, la descente du gradient comporte des boucles de longueur constantes à l'intérieur d'une boucle d'itération locale de longueur pratiquement invariable.

La version multiélément de Nelder-Mead change la donne. En effet, elle permet une plus grande uniformité entre les fils d'exécution en multipliant les fils qui traitent un même voisinage élémentaire. Puisque les conditions des branchements conditionnels de Nelder-Mead sont basées sur la qualité des voisinages élémentaires, tous les fils qui travaillent en coopération sur un même voisinage empruntent les mêmes branches, diminuant ainsi la divergence entre les fils d'exécution.

L'implémentation CUDA de la version multiélément de Nelder-Mead accélère suffisamment l'algorithme pour qu'il devienne plus rapide que la descente du gradient, devenant par le fait même l'implémentation la plus rapide de tous les algorithmes d'optimisation locale. On jumellera l'implémentation CUDA de la version multiélément de Nelder-Mead avec l'échantillonnage de la métrique par texture pour former le pipeline optimal d'adaptation sur GPU.

5.5 Comparaison des pipelines CPU et GPU

Nos deux pipelines d'adaptation optimaux sont donc maintenant fixés. Le pipeline CPU échantillonne la fonction métrique par recherche locale dans un maillage de fond. Le pipeline GPU échantillonne quant à lui la fonction métrique à l'aide de deux textures 3D. Nos tests ont démontré que les deux techniques offrent des précisions comparables pour des fonctions métriques de difficulté modérée. Finalement, en termes de technique de déplacement de nœuds, les deux pipelines s'accordent pour utiliser l'algorithme de Nelder-Mead. Entre les deux versions disponibles (standard et multiélément) et entre les deux implémentations (GLSL et CUDA), on utilisera l'implémentation CUDA de la version multiélément pour le pipeline GPU.

Ces pipelines ont été choisis en fonction de deux maillages : un cube divisé en tétraèdres et un cube divisé en hexaèdres. Cependant, on ne connaît pas la relation entre la taille d'un maillage et le temps d'exécution. Un premier test tentera d'illustrer cette relation pour les deux pipelines choisis.

On aimera également savoir comment les pipelines se comporteront en pratique sur des cas réels. Cela implique d'étudier des maillages issus de travaux d'ingénierie. Pour ces cas, on échangera la métrique spécifiée analytique contre la vraie métrique construite à partir de l'estimation a posteriori de l'erreur de discrétisation pour comparer les deux pipelines. Ces tests seront exécutés sur plusieurs configurations matérielles afin d'évaluer l'impact du CPU et de la carte graphique utilisé sur les accélérations obtenues.

5.5.1 Croissance du temps de calcul

Examiner le comportement des pipelines mis à l'échelle nous permet d'évaluer la capacité des algorithmes à traiter des problèmes de plus en plus volumineux. Puisque les algorithmes de déplacement de nœuds opèrent sur un grand ensemble de données ayant peu d'interactions entre elles, on s'attend à ce que le temps de calcul soit directement proportionnel à la taille des maillages, surtout pour le pipeline CPU. Il est moins clair toutefois que le pipeline GPU se comporte de la même façon, notamment pour les maillages de petite taille. En effet, les

nœuds d'un maillage sont déjà partitionnés en ensembles de nœuds indépendants puis divisés en blocs de calcul. Plus le nombre de nœuds est petit, plus le taux de blocs partiellement remplis augmente et, incidemment, plus les coûts de gestion des noyaux de calcul et plus la latence des copies mémoires deviennent importants face au temps de calcul.

Le test mis en place consiste à adapter par déplacement de nœuds des maillages de plus en plus volumineux. Huit maillages seront adaptés successivement de tailles $N_i = 10e4 \times 2^i$, $i \in [1, 8]$ étant l'indice du maillage. On teste les implémentations séquentielle et parallèle du pipeline CPU ainsi que les implémentations GLSL et CUDA du pipeline GPU. La métrique spécifiée est encore une fois la métrique analytique sinusoïdale qui simule des chocs perpendiculaires au vecteur $\vec{u}$. Le facteur de compression A est maintenu à 8 tandis que le facteur de mise à l'échelle est ajusté afin d'obtenir les tailles voulues pendant la phase préliminaire de modifications topologiques. Les tailles des maillages ne correspondent pas exactement aux tailles espérées N_i, mais respectent une marge d'erreur de 1.5%. La croissance des temps d'exécution est illustrée à la figure 5.14. Les temps sont présentés au tableau 5.18 en secondes.

Comme prévu, on observe une progression linéaire du temps de calcul en fonction de la taille des maillages pour les implémentations séquentielle et parallèle. L'implémentation CUDA tend vers une progression linéaire pour les maillages de grande taille, mais on voit bien que les maillages de petite taille font légèrement dévier cette progression en exprimant une relation sous-linéaire. Il semble donc qu'en dessous d'une dizaine de milliers de nœuds, les coûts associés à la gestion de noyaux de calculs sur le GPU sont significatifs. L'implémentation GLSL ne suit pas exactement la même tendance. Elle démontre la même progression sous-linéaire pour les maillages de petite taille, mais atteint rapidement une progression linéaire entre 20 000 et 160 000 nœuds avant de voir son ordre de croissance grimper abruptement en s'approchant du million de nœuds. En fait, il est même arrivé que le programme plante lorsque l'implémentation était lancée sur des maillages de cette taille. Il est difficile d'expliquer le

Tableau 5.18 Ordre de croissance du temps de calcul en fonction de la taille du maillage

Tailles	Séquentiel	Parallèle	GLSL	CUDA
10000	15.22	3.44	1.13	0.72
20 000	30.77	6.60	1.89	1.04
40 000	64.14	13.35	3.59	1.66
80 000	136.26	27.22	7.03	2.86
160 000	269.27	55.30	14.39	5.34
320 000	552.15	112.78	30.39	10.43
640 000	1123.16	229.38	75.85	20.74
1 280 000	2250.53	467.18	457.70	42.15

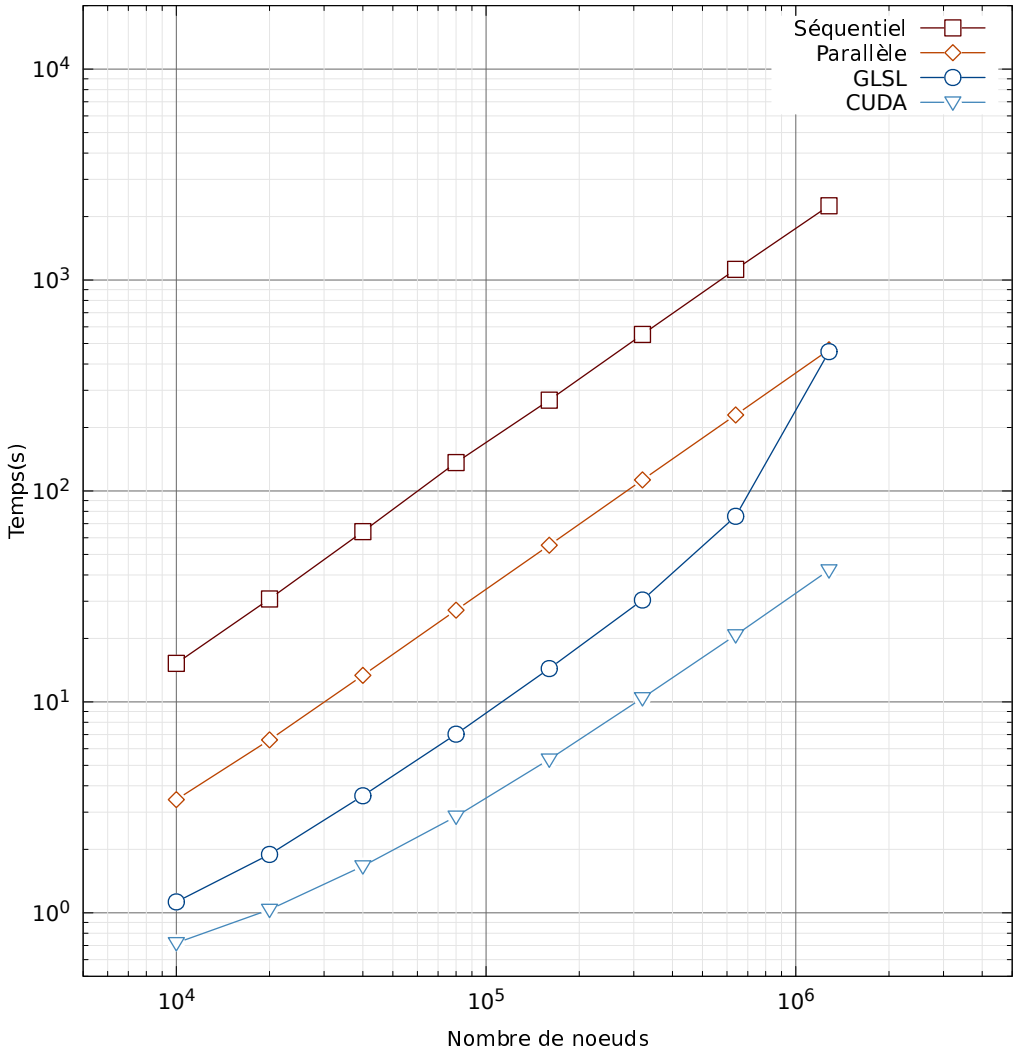


Figure 5.14 Ordre de croissance du temps de calcul en fonction de la taille du maillage

comportement de l'implémentation GLSL pour les maillages de grande taille. Profiler le noyau de calcul nous aiderait probablement à cerner la cause de ce problème, mais, contrairement à CUDA, il n'existe aucun outil pour déboguer les noyaux de calcul GLSL pour le moment. L'implémentation parallèle offre des taux d'accélération de 4,9 contre des taux de 52,9 pour l'implémentation CUDA sur des maillages de grandes tailles.

Puisque les maillages comportant moins de dix mille nœuds sont rares en analyse numérique, les coûts de gestion ne représentent pas un problème réel. De plus, l'implémentation CUDA garde une importante avance sur tout le spectre des tailles étudiées. L'analyse de mise à l'échelle confirme bien que l'implémentation CUDA de Nelder-Mead multiélément jumelée à l'échantillonnage de la métrique spécifiée par texture constitue le pipeline optimal d'adaptation de maillage, peu importe la quantité de nœuds à déplacer. Il ne reste plus qu'à mettre à l'épreuve ce pipeline dans des contextes réels.

5.5.2 Cas réel

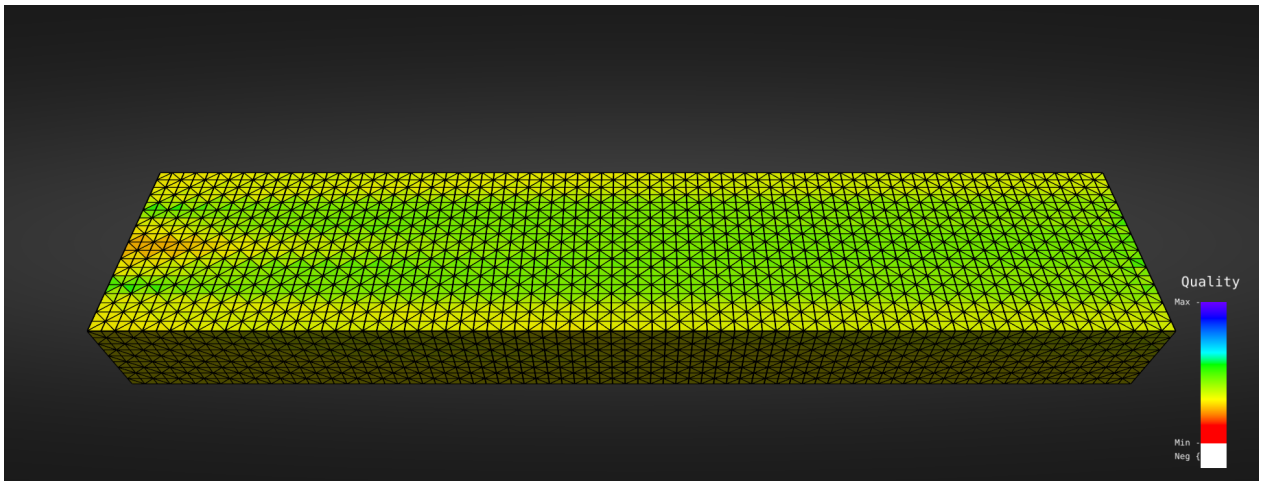
Les cas de test réels sont fournis au format Pirate (extension .pie). Puisque le logiciel d'adaptation ne permet pas de lire la géométrie des frontières dans ce format, les nœuds qui reposent sur la frontière resteront immobiles. En d'autres mots, seuls les nœuds sous-surfaciques et intérieurs seront adaptés par déplacement de nœuds. Les maillages et leur métrique spécifiée ont été recentrés et mis à l'échelle pour entrer dans la boîte englobante de coins $(-1, -1, -1)$ à $(1, 1, 1)$. De cette façon, on maximise la précision des nombres à virgule flottante, ce qui est particulièrement utile pour les versions GPU qui travaillent en simple précision.

Le premier cas réel, nommé *Jet dans une boîte,* représente un jet à l'intérieur d'une boîte rectangulaire de 32 000 nœuds et 142 595 tétraèdres. Avant adaptation, la qualité du pire élément est de 0.206 et la moyenne harmonique est de 0.383. Après 300 itérations de Nelder-Mead, on obtient une moyenne de 0.400. Les implémentations CPU gardent la qualité minimale intouchée tandis que les implémentations GPU la font monter à 0.225. La résolution par défaut des textures est $93 \times 19 \times 19$ cellules. Afin d'obtenir les mêmes moyennes de qualité entre l'échantillonnage par recherche locale et par texture, la résolution a été augmentée à $117 \times 23 \times 23$, ce qui correspond à 1,85 fois plus de cellules que la résolution par défaut. L'intérieur des maillages avant et après adaptation sont présentés à la figure 5.15. La disposition des éléments dans le maillage adapté nous fait deviner l'entrée du jet à la gauche de la boîte.

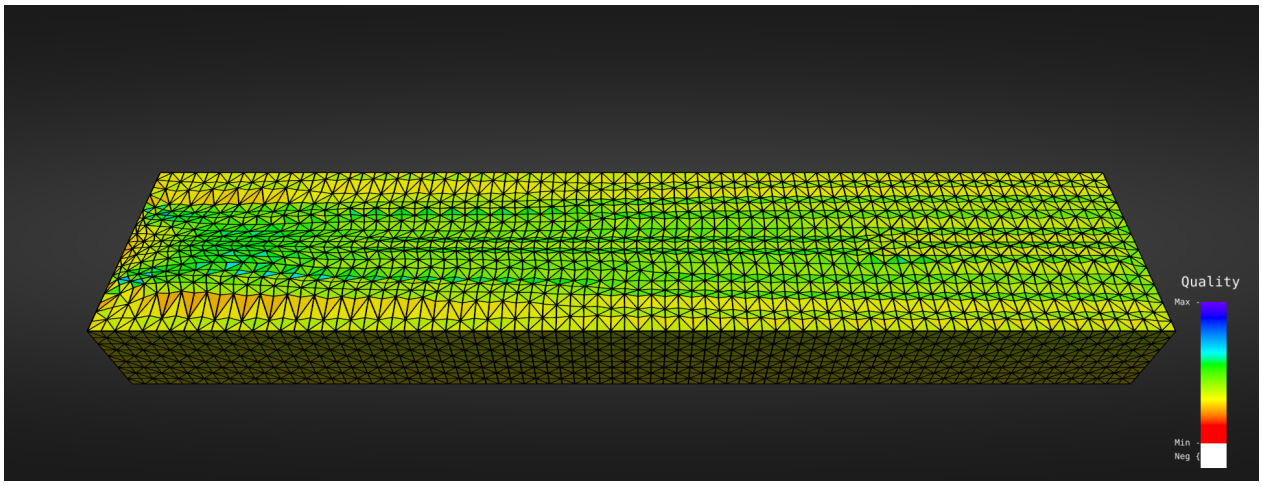
La première configuration matérielle testée n'est nulle autre que la configuration utilisée pour exécuter tous les tests synthétiques de ce chapitre. Les temps obtenus concordent avec les résultats des tests précédents. Cette configuration obtient un taux d'accélération approchant les 5x pour la version parallèle et un taux approchant les 50x pour la version CUDA. Encore

Tableau 5.19 Temps d'adaptation du *Jet dans une boîte* à l'aide des pipelines optimaux

Configurations	Temps (ms)				Accélérations		
	Séquentiel	Parallèle	GLSL	CUDA	Parallèle	GLSL	CUDA
Desktop NVIDIA	945.38	200.36	48.90	20.92	4.7	19.3	45.2
Laptop NVIDIA	1663.14	370.55	∅	95.18	4.5	∅	17.5



(a) Initial



(b) Adapté

Figure 5.15 Maillage initial et adapté du jet dans la boîte

une fois, l'implémentation GLSL est plus lente que l'implémentation CUDA.

La deuxième configuration matérielle est un ordinateur portable. Puisque cette configuration possède le même type de processeur que la première, soit un processeur quadricœur avec hyperthread, l'implémentation parallèle offre le même taux d'accélération. Toutefois, son processeur est un peu moins puissant. Elle prend donc 1,75 fois plus de temps pour compléter les 300 itérations. L'implémentation CUDA présente un taux d'accélération inférieur à la première configuration. Cette différence s'explique facilement par la différence de gamme des cartes graphiques utilisées. Malgré tout, l'implémentation CUDA est 17,5 fois plus rapide que l'implémentation sérielle et 3,9 fois plus rapide que la version parallèle. Donc, l'utilisation d'un GPU représente encore une fois un avantage appréciable sur cette configuration.

Notez qu'aucun temps n'a été fourni pour l'implémentation GLSL pour la deuxième configuration matérielle. La raison est qu'un défaut logiciel perdure depuis un an dans les pilotes d'OpenGL fourni par NVIDIA sur Linux. Au-delà de la version 367 du pilote, l'application termine abruptement à l'édition des liens des nuanceurs de calcul les plus volumineux. Le problème est que la librairie CUDA est livrée avec sa propre version des pilotes et qu'il est extrêmement compliqué d'effectuer une régression tout en préservant la stabilité du système. Cette régression a été complétée avec succès sur la première configuration matérielle. Malheureusement, cette régression cause des boucles de réinitialisation vidéo au quatre secondes sur la deuxième, rendant le système pratiquement inutilisable. Le défaut logiciel du pilote est connu des développeurs. Il semblait avoir été réglé il y a quelques mois pour certaines personnes, mais le problème perdure encore aujourd'hui avec la dernière version du pilote au lancement des implémentations GLSL. Malgré sa grande portabilité, l'instabilité de l'implémentation GLSL est un incitatif de plus pour ne retenir que l'implémentation CUDA.

Le présent cas de test confirme les conclusions tirées des tests synthétiques. L'implémentation CUDA offre de meilleurs taux d'accélération que l'implémentation parallèle sur CPU, et ce pour des configurations matérielles très différentes. De plus, on voit que l'implémentation GLSL est plus rapide que l'implémentation parallèle, mais au moins deux fois plus lente que l'implémentation CUDA. Dans le cas où l'on souhaiterait réduire le temps d'adaptation de maillage en tirant profit d'un GPU, il est conseillé d'intégrer la version multiélément de Nelder-Mead écrite en CUDA.

CHAPITRE 6 CONCLUSION

La présente recherche a permis le développement d'un banc de test pour la conception, la vérification et la comparaison d'algorithmes d'adaptation de maillage. L'objectif principal consistait à déterminer s'il est possible d'accélérer l'exécution d'algorithmes de déplacement de nœuds sur GPU pour l'adaptation de maillage avec métrique riemannienne. Il a été démontré qu'il est non seulement possible de développer de tels algorithmes, mais que ceux-ci produisent des maillages d'aussi bonne qualité que leurs versions parallèles sur CPU.

On fera un court retour sur les travaux réalisés tout en mettant l'accent sur les contributions de la recherche. Puis, on énoncera les limites et les contraintes de la solution proposée. Finalement, quelques travaux futurs seront proposés qui permettraient d'intégrer la solution actuelle à un logiciel complet d'adaptation de maillage en mécanique des fluides numérique.

6.1 Synthèse des travaux

On a vu au chapitre 2 l'ensemble des concepts essentiels en adaptation de maillage : le contrôle de l'erreur, l'utilisation d'une métrique riemannienne, les mesures de qualité, les algorithmes de déplacement de nœuds et les algorithmes de modifications topologiques. Le contrôle de l'erreur par le lemme de Céa est un concept mathématique qui nous intéresse principalement par son implication sur les mesures de qualité. Les mesures de qualité tentent d'évaluer l'impact d'un élément sur l'erreur d'interpolation en fonction de sa taille et de sa forme. On sait, par le lemme de Céa, que l'erreur d'interpolation borne l'erreur d'approximation et donc qu'en réduisant la première on réduit également la deuxième. Un grand nombre de mesures de qualité ont été développées au fil du temps. Cependant, seule la mesure de la conformité à la métrique permet d'évaluer précisément la qualité d'un élément en fonction de sa forme et de sa taille dans une métrique riemannienne.

Depuis une solution approchée préliminaire, la manière standard de fournir une fonction métrique est de stocker les tenseurs métriques aux nœuds du maillage préliminaire. On obtient ainsi un maillage de fond sur lequel il est possible d'échantillonner la fonction métrique par recherche locale. Cette manière de fournir et d'échantillonner la fonction métrique fonctionne parfaitement sur le CPU et il n'y a aucune raison de la remplacer. Toutefois, cette représentation n'est pas appropriée au GPU. Elle dégrade les performances à un point tel que les algorithmes de déplacements n'arrivent même pas à terminer.

La première contribution de la présente recherche est la conception d'une nouvelle repré-

sentation de la fonction métrique. Le maillage de fond est discrétisé sous forme de grille uniforme, puis stockée en tant que texture sur le GPU. Chaque cellule de la grille représente un tenseur métrique. La grille est initialisée en échantillonnant la métrique dans le maillage de fond au centre des cellules. Puis, pour éviter d'accéder à des cellules non initialisées au cours du processus d'adaptation, on initialise une couche de cellules supplémentaire autour du domaine. Bien que Rokos et al. (2011) aient proposé l'utilisation de texture 2D pour adapter des maillages bidimensionnels, ils n'ont pas présenté de technique robuste pour générer de telles textures et, de manière plus critique, ils n'ont pas vérifié si cette technique offrait la même précision qu'un maillage de fond. Or, les tests de précision des techniques d'échantillonnage nous ont démontré que la précision des textures n'était pas garantie, bien qu'en ajustant avec la résolution il soit possible de rejoindre la précision du maillage de fond.

Après la sélection des techniques d'échantillonnage optimales sur CPU et sur GPU, plusieurs algorithmes de déplacement de nœuds ont été étudiés sur les deux processeurs. Au sujet du lissage laplacien, on a vu que, malgré sa popularité, ce type d'algorithme est moins efficace que les algorithmes d'optimisation locale en adaptation de maillage, bien qu'il soit de un à deux ordres de grandeur plus rapide qu'eux et qu'il possède des taux d'accélération inégalés sur GPU. La deuxième contribution de la présente recherche est la généralisation du lissage laplacien aux espaces riemanniens. Bien que l'algorithme ait été écarté des pipelines optimaux, cette contribution pourrait être utile dans le cas où plusieurs algorithmes de déplacement de nœuds seraient enchaînés afin de minimiser le temps de calcul et maximiser le gain total de qualité.

Quant aux algorithmes d'optimisation locale, ils possèdent tous des efficacités équivalentes pour augmenter la conformité d'un maillage à une métrique riemannienne. Nelder-Mead s'est toutefois démarqué par sa vitesse et il a donc été choisi comme algorithme de déplacement optimal pour les pipelines d'adaptation CPU et GPU. Pour éviter de complexifier les noyaux de calcul, le déplacement des nœuds frontières est toujours fait sur le CPU. Cela diminue la divergence des fils d'exécution GPU sans compter que le niveau de parallélisme est augmenté par la distribution du travail entre deux processeurs.

La troisième, et la plus importante, contribution de la présente recherche est l'élaboration de nouveaux espaces de parallélisation. Il est naturel de paralléliser le déplacement de nœuds sur CPU en distribuant le traitement des nœuds du maillage entre plusieurs fils d'exécution. Bien qu'aucun article présenté à la section 2.7 ne soit explicite à ce sujet, on suppose que toutes les recherches sur la parallélisation d'algorithmes de déplacement de nœuds sur GPU se sont bornées à cet espace. Cependant, les GPU sont des processeurs massivement parallèles et offrent la possibilité d'exécuter un grand nombre de fils d'exécution dont le comportement

est uniforme (c.-à-d. qui exécute la même suite d'instructions). L'exploration des espaces de parallélisation par élément et par position a permis le développement d'algorithmes offrant un plus grand niveau de parallélisme et dont le comportement des fils d'exécution est plus uniforme. Notamment, la version multiaxe de la descente du gradient et la version multiélément de Nelder-Mead sont deux à trois fois plus rapides que leur version standard parallélisée seulement dans l'espace des nœuds sur GPU. Le pipeline optimal d'adaptation sur GPU échantillonne la métrique par texture et déplace les nœuds par la version multiélément de Nelder-Mead écrit en CUDA. Ce pipeline est en moyenne 50 fois plus rapide que l'implémentation séquentielle et 10 fois plus rapide que l'implémentation parallèle sur la principale configuration matérielle testée. La configuration matérielle secondaire nous rappelle toutefois que ces taux d'accélération dépendent fortement du CPU et du GPU utilisés.

La dernière contribution concerne la nature des maillages adaptés sur GPU. Toutes les recherches jusqu'à présent se sont penchées sur des maillages triangulaires en 2D ou tétraédriques en 3D. En mécanique des fluides numérique, il est fréquent de rencontrer des couches d'hexaèdres sur les parois. On retrouve également des prismes dans les maillages générés par l'extrusion de maillages triangulaires. C'est pourquoi il a été jugé important de supporter ces types d'éléments dans le logiciel d'adaptation. Supporter 3 types d'éléments plutôt qu'un ajoute de la complexité à l'infrastructure GPU de l'adaptateur et augmente les chances de divergence dans les noyaux de calcul, ce qui peut réduire leurs performances. Mais ces trois types d'éléments sont rencontrés régulièrement en mécanique des fluides numérique et ne pas supporter chacun d'eux limiterait la pertinence de la solution proposée.

Un objectif secondaire de la présente recherche était de comparer différentes implémentations GPU pour connaître leurs forces et leurs faiblesses. On s'est limité à OpenGL (GLSL étant son langage de programmation pour les noyaux) et CUDA. Un premier avantage d'écrire les noyaux de calculs en GLSL est qu'OpenGL est la librairie graphique utilisée pour visualiser les maillages dans le logiciel d'adaptation. Un autre avantage est qu'OpenGL peut utiliser les cartes graphiques des deux fabricants majeurs (NVIDIA et AMD) contrairement à CUDA qui, étant une technologie propriétaire, ne peut s'exécuter que sur les cartes NVIDIA. Par contre, CUDA est un langage plus mature que GLSL et possède un environnement de développement plus riche que ce dernier. On a accès entre autres à des profileurs et des débogueurs. Bien que CUDA se soit révélé être plus performant que GLSL pour exécuter le pipeline d'adaptation optimal, on remarquera que d'autres facteurs encourage l'utilisation de CUDA. Outre l'impossibilité de profiler et de déboguer les noyaux de calcul GLSL, on a remarqué que ceux-ci n'étaient pas fiables. Les temps d'exécution présentent de fortes variations d'un lancement à l'autre, certains cas de test font planter le système et le test de mise à l'échelle montre leur inefficacité à traiter des maillages volumineux. On peut également mentionner le bogue qui

perdure depuis un an dans les pilotes d'NVIDIA sur linux et qui nous a empêché de tester l'adaptation du jet dans la boîte sur l'ordinateur portable en GLSL.

En définitive, le pipeline d'adaptation optimal est composé de la conformité à la métrique comme évaluateur, de l'échantillonnage par texture et du déplacement de nœuds par Nelder-Mead. Ce pipeline est exécuté sur GPU, est écrit en CUDA et parallélise l'exécution selon l'espace des nœuds et des éléments (version multiélément). La grille de calcul est divisée en bloc de 32 fils d'exécution pour l'évaluation et le lissage du maillage.

6.2 Limitations de la solution proposée

La première limite actuelle du logiciel d'adaptation concerne la représentation des nombres en virgule flottante. Tous les nombres, sauf quelques résultats intermédiaires de calcul, sont représentés en simple précision dans les implémentations GPU. La raison est que les cartes graphiques utilisées pour produire les résultats possèdent 24 fois moins d'unités double précision que simple. Bien qu'il existe des accélérateurs de calcul possédant de meilleurs ratios simple/double précision, ces cartes coûtent plus cher et sont destinées à un marché de calcul haute performance. Elles sont donc moins facilement accessibles que les cartes graphiques produites pour la masse, comme les GeForce^{MC} et les Radeon^{MC}. Pour exécuter le logiciel d'adaptation sur l'un de ces accélérateurs graphiques, il suffirait de remplacer tous les nombres en simple précision par des nombres en double précision et l'on bénéficierait directement d'une précision de calcul supérieure. Malheureusement, puisque la majorité des utilisateurs potentiels ne possède qu'une carte graphique de masse, ils devront se contenter de la simple précision. Ceci limite la taille des maillages adaptables et proscrit la présence de détails géométriques fins sur la frontière.

On se rappellera que les opérations topologiques et les opérations géométriques sont deux classes d'opérations complémentaires en adaptation de maillage. L'accent a été mis sur les opérations géométriques, c'est-à-dire les algorithmes de déplacement de nœuds, plutôt que les opérations topologiques dans ce mémoire. Les algorithmes de modifications topologiques étant difficilement parallélisables sur GPU, les algorithmes de déplacement de nœuds constituaient la seule voie praticable pour accélérer le processus entier d'adaptation de maillage à l'aide de ce type de processeur. C'est pourquoi les résultats se limitent à l'étude des efficacités relatives ainsi qu'aux vitesses relatives des algorithmes de déplacement de nœuds.

Toutefois, les opérations topologiques et géométriques sont généralement entrelacées par l'adaptateur. Par exemple, lors d'un effondrement d'arête, les nœuds fusionnés et leurs voisins sont déplacés afin d'obtenir la meilleure configuration locale avant de passer à la prochaine

opération topologique. Actuellement, le logiciel procède par lots d'opérations du même type. Plusieurs itérations globales d'opérations topologiques sont exécutées avant de passer aux itérations globales de déplacement de nœuds. Cette manière de procéder ralentit le taux de convergence du processus ainsi que la capacité des opérations topologiques à régulariser la topologie du maillage.

Un des avantages du modèle entrelacé face au modèle en lots est sa capacité à se concentrer sur les éléments de faible qualité. La qualité du pire élément d'un maillage nous intéresse puisque l'on sait qu'il suffit d'un petit nombre d'éléments dégénérés pour compromettre la solution sur tout le domaine. Le modèle d'opérations en lots consacre un temps égal à chaque nœud, qu'il soit entouré d'éléments de mauvaise qualité ou pas. Au contraire, le modèle entrelacé peut tester plusieurs opérations topologiques sur un même groupe d'éléments, déplacer leurs nœuds, visiter les environs avant de revenir sur les éléments problématiques pour tenter de les débloquer. Des fils de priorités sont généralement utilisés pour implémenter ce modèle afin de classer les nœuds, les arêtes et les éléments selon leur degré de conformité à la métrique. Prioriser les nœuds en fonction de la qualité de leur voisinage élémentaire est incompatible avec le modèle en lots et il n'est donc pas possible d'octroyer plus de temps à certains nœuds dans le logiciel d'adaptation actuel.

6.3 Améliorations futures

Le modèle d'opérations en lots utilisé par le logiciel d'adaptation n'est pas simplement un défaut de conception. Il provient du modèle de parallélisation choisi. Avant d'être déplacés parallèlement, les nœuds du maillage doivent être partitionnés en ensembles de nœuds indépendants. Puisque les opérations topologiques modifient la connectivité des nœuds, elles changent par conséquent le partitionnement des nœuds localement, et possiblement globalement. De plus, pour pouvoir paralléliser un modèle d'opérations entrelacées entièrement, il faudrait développer une méthode pour partitionner les nœuds non seulement en fonction des opérations géométriques, mais également en fonction des opérations topologiques. D'Amato and Vénere (2013) offrent une piste de solution sous forme de *clusters* pour remplacer les ensembles de nœuds indépendants. Un *cluster* est un petit groupe d'éléments réservés pour une opération, qu'elle soit géométrique ou topologique. Plusieurs *clusters* peuvent être traités simultanément tout en garantissant la validité des résultats. Comme les ensembles de nœuds indépendants, l'intersection de tous les *clusters* manipulés simultanément doit être vide.

Un ensemble de nœuds indépendants est en quelque sorte une représentation compacte de tous les *clusters* qui peuvent être manipulés simultanément par un algorithme de déplacement de nœuds. Inclure les opérations topologiques au modèle de traitement parallèle demanderait de

généraliser le concept d'ensemble indépendant au concept de *cluster* pour pouvoir manipuler non seulement des voisinages élémentaires, mais toute forme de régions connexes dans un maillage. De cette façon, il serait possible de passer du modèle en lots au modèle entrelacé et ainsi octroyer plus de temps aux éléments de mauvaise qualité et sauver du temps sur les éléments de bonne qualité. Par contre, ce changement impliquerait une réécriture majeure du logiciel d'adaptation. Les lisseurs et les topologistes n'ont pas du tout été conçus pour travailler aussi étroitement.

Dans un autre ordre d'idée, la quantité de mémoire disponible sur le GPU est un facteur qui limite actuellement la taille des maillages pouvant être adaptés. Les cartes graphiques communes ne possèdent généralement que 2 à 8 Go de mémoire dédiée, juste assez pour adapter des maillages de quelques millions de nœuds. Même les accélérateurs graphiques comme les Tesla ne possèdent pas plus de 12 Go. Il existe cependant des techniques pour adapter des maillages trop volumineux pour entrer en mémoire principale. Le maillage doit d'abord être partitionné en régions de tailles raisonnables. Les régions sont tour à tour chargées en mémoire principale, adaptées puis réenregistrées sur le disque. Les surfaces à la jonction des régions doivent être gardées intactes pendant le traitement des régions. Une étape finale extrait les éléments des jonctions pour qu'ils soient adaptés à leur tour. La même technique peut être utilisée pour accélérer le déplacement de nœuds des maillages trop volumineux pour entrer d'un coup sur le GPU.

Parce que l'implémentation GLSL du pipeline optimal d'adaptation est peu fiable et difficile à déboguer, il est recommandé de n'utiliser que l'implémentation CUDA, ce qui limite la portée de la solution aux cartes graphiques fabriquées par NVIDIA. Cependant, d'autres librairies multiplateformes de programmation GPU existent qui ressemblent à OpenGL. Actuellement, la librairie la plus intéressante est Vulkan. Au moment d'écrire ces lignes, la première spécification de Vulkan n'a été publiée qu'il y a quelques mois, mais elle semble bénéficier de toute l'expérience acquise par la communauté GPGPU de la dernière décennie.

RÉFÉRENCES

F. J. Bossen et P. S. Heckbert, “A pliant method for anisotropic mesh generation”, dans *5th Intl. Meshing Roundtable*. Citeseer, 1996, pp. 63–74.

J. Cea, “Approximation variationnelle des problèmes aux limites.” *Ann. Inst. Fourier*, vol. 14, no. 2, pp. 345–444, 1964. DOI : 10.5802/aif.181

Z. Cheng, E. Shaffer, R. Yeh, G. Zagaris, et L. Olson, “Efficient parallel optimization of volume meshes on heterogeneous computing systems”, *Engineering with Computers*, pp. 1–10, 2015. DOI : 10.1007/s00366-014-0393-7. En ligne : http://dx.doi.org/10.1007/s00366-014-0393-7

A. R. Conn, K. Scheinberg, et L. N. Vicente, *Introduction to derivative-free optimization*. Siam, 2009, vol. 8.

A. Corrigan et R. Löhner, “Semi-automatic porting of a large-scale CFD code to multi-graphics processing unit clusters”, *International Journal for Numerical Methods in Fluids*, vol. 69, no. 11, pp. 1786–1796, aug 2011. DOI : 10.1002/fld.2664. En ligne : http://dx.doi.org/10.1002/fld.2664

S. Dahal et T. S. Newman, “Efficient, gpu-based 2d mesh smoothing”, dans *IEEE SOUTHEASTCON 2014*, March 2014, pp. 1–7. DOI : 10.1109/SECON.2014.6950720

J. D’Amato et M. Vénere, “A CPU–GPU framework for optimizing the quality of large meshes”, *Journal of Parallel and Distributed Computing*, vol. 73, no. 8, pp. 1127–1134, aug 2013. DOI : 10.1016/j.jpdc.2013.03.007. En ligne : http://dx.doi.org/10.1016/j.jpdc.2013.03.007

J. Dompierre, M.-G. Vallet, P. Labbé, et F. Guibault, “An analysis of simplex shape measures for anisotropic meshes”, *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 48-49, pp. 4895–4914, nov 2005. DOI : 10.1016/j.cma.2004.11.018. En ligne : http://dx.doi.org/10.1016/j.cma.2004.11.018

J. Dompierre, P. Labbé, M.-G. Vallet, et R. Camarero, “How to subdivide pyramids, prisms, and hexahedra into tetrahedra.” dans *IMR*, 1999, pp. 195–204.

L. Freitag, M. Jones, et P. Plassmann, “A parallel algorithm for mesh smoothing”, *SIAM Journal on Scientific Computing*, vol. 20, no. 6, pp. 2023–2040, 1999.

L. A. Freitag et P. M. Knupp, “Tetrahedral mesh improvement via optimization of the element condition number”, *International Journal for Numerical Methods in Engineering*, vol. 53, no. 6, pp. 1377–1391, 2002.

L. A. Freitag et C. Ollivier-Gooch, “Tetrahedral mesh improvement using swapping and smoothing”, *International Journal for Numerical Methods in Engineering*, vol. 40, no. 21, pp. 3979–4002, 1997. DOI : 10.1002/(SICI)1097-0207(19971115)40:21<3979::AID-NME251>3.0.CO;2-9. En ligne : http://dx.doi.org/10.1002/(SICI)1097-0207(19971115)40:21<3979::AID-NME251>3.0.CO;2-9

P. Frey et P. George, *Maillages : applications aux éléments finis*. Hermès Science Publications, 1999. En ligne : https://books.google.ca/books?id=RLJ9AAAACAAJ

L. R. Herrmann, “Laplacian-isoparametric grid generation scheme”, *Journal of the Engineering Mechanics Division*, vol. 102, no. 5, pp. 749–907, 1976.

B. M. Klingner et J. R. Shewchuk, *Aggressive Tetrahedral Mesh Improvement*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, pp. 3–23. DOI : 10.1007/978-3-540-75103-8_1. En ligne : http://dx.doi.org/10.1007/978-3-540-75103-8_1

P. Knupp, “Matrix norms and the condition number : A general framework to improvemesh quality via node-movement, 8th international meshing roundtable”, *Lake Tahoe, pp13-22*, 1999.

P. M. Knupp, “Algebraic mesh quality metrics”, *SIAM Journal on Scientific Computing*, vol. 23, no. 1, pp. 193–218, 2001. DOI : 10.1137/S1064827500371499. En ligne : http://dx.doi.org/10.1137/S1064827500371499

P. Labbé, J. Dompierre, M.-G. Vallet, F. Guibault, et J.-Y. Trépanier, “A universal measure of the conformity of a mesh with respect to an anisotropic metric field”, *International Journal for Numerical Methods in Engineering*, vol. 61, no. 15, pp. 2675–2695, oct 2004. DOI : 10.1002/nme.1178. En ligne : http://dx.doi.org/10.1002/nme.1178

P. Labbé, J. Dompierre, M.-G. Vallet, et F. Guibault, “Verification of three-dimensional anisotropic adaptive processes”, *International Journal for Numerical Methods in Engineering*, vol. 88, no. 4, pp. 350–369, mar 2011. DOI : 10.1002/nme.3178. En ligne : http://dx.doi.org/10.1002/nme.3178

A. Liu et B. Joe, “Relationship between tetrahedron shape measures”, *BIT*,

vol. 34, no. 2, pp. 268–287, jun 1994. DOI : 10.1007/bf01955874. En ligne : http://dx.doi.org/10.1007/BF01955874

——, “On the shape of tetrahedra from bisection”, *Mathematics of Computation*, vol. 63, no. 207, pp. 141–141, sep 1994. DOI : 10.1090/s0025-5718-1994-1240660-4. En ligne : http://dx.doi.org/10.1090/S0025-5718-1994-1240660-4

D. S. Lo, *Finite Element Mesh Generation*. CRC Press, 2015.

S. Lo, “Volume discretization into tetrahedra-ii. 3d triangulation by advancing front approach”, *Computers and Structures*, 1991. En ligne : http://hdl.handle.net/10722/149956

——, “Optimization of tetrahedral meshes based on element shape measures”, *Computers & structures*, vol. 63, no. 5, pp. 951–961, 1997.

G. Mei, J. C. Tipper, et N. Xu, “A generic paradigm for accelerating laplacian-based mesh smoothing on the GPU”, *Arab J Sci Eng*, vol. 39, no. 11, pp. 7907–7921, oct 2014. DOI : 10.1007/s13369-014-1406-y. En ligne : http://dx.doi.org/10.1007/s13369-014-1406-y

H. Nguyen, *Gpu Gems 3*, 1er éd. Addison-Wesley Professional, 2007.

NVIDIA, “Nvidia’s next generation cuda™ compute architecture : Kepler™ gk110”, Rapp. tech., 2012.

J. Park et S. M. Shontz, “Two derivative-free optimization algorithms for mesh quality improvement”, *Procedia Computer Science*, vol. 1, no. 1, pp. 387–396, 2010.

G. Rokos, G. Gorman, et P. H. J. Kelly, *Accelerating Anisotropic Mesh Adaptivity on nVIDIA’s CUDA Using Texture Interpolation*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, pp. 387–398. DOI : 10.1007/978-3-642-23397-5_38. En ligne : http://dx.doi.org/10.1007/978-3-642-23397-5_38

J. Shewchuk, “What is a good linear finite element ? interpolation, conditioning, anisotropy, and quality measures (preprint)”, *University of California at Berkeley*, vol. 73, p. 12, 2002.

J. R. Shewchuk, “Two discrete optimization algorithms for the topological improvement of tetrahedral meshes”, *Unpublished manuscript*, vol. 65, 2002.

S. Singer et J. Nelder, “Nelder-mead algorithm”, *Scholarpedia*, vol. 4, no. 7, p. 2928, 2009. DOI : 10.4249/scholarpedia.2928. En ligne : http://dx.doi.org/10.4249/scholarpedia.2928

D. J. Struik, *De impletione loci, Nieuw Archief voor Wiskund*, 1925.

M.-G. Vallet, “Génération de maillages éléments finis anisotropes et adaptatifs”, Thèse de doctorat, 1992, thèse de doctorat dirigée par Pironneua, Olivier Mathématiques Paris 6 1992. En ligne : http://www.theses.fr/1992PA066625

R. Wang, S. Gao, Z. Zheng, et J. Chen, “Frame field guided topological improvement for hex mesh using sheet operations”, *Procedia Engineering*, vol. 163, pp. 276–288, 2016. DOI : 10.1016/j.proeng.2016.11.060. En ligne : https://doi.org/10.1016/j.proeng.2016.11.060

ANNEXE A Descriptions de classe

Gestionnaires

GpuMeshCharacter

Gestionnaire d'application. Contrôle les interactions entre l'interface usager, les pipelines d'adaptation et la visualisation des maillages.

MastersTestSuite

Gestionnaire de cas de test. Exécute tous les tests présentés à l'analyse des résultats.

Structures de données

DataStructures/Mesh

Implémentation de base pour le stockage et la manipulation du maillage sur le CPU.

DataStructures/GpuMesh

Classe dérivée de *Mesh* pour le stockage et la manipulation du maillage sur le GPU.

DataStructures/MeshCrew

Assemblage du sous-pipeline d'échantillonnage, de mesure et d'évaluation.

DataStructures/NodeGroups

Divers partitionnement du tableau de nœuds pour l'exécution des algorithmes de déplacement de nœuds en fonction du processeur utilisé.

Échantillonnage de la métrique

Samplers/UniformSampler

Échantillonneur pour la métrique uniforme. Seul échantillonneur qui travaille de pair avec le mesureur indépendant de la métrique.

Samplers/AnalyticSampler

Échantillonneur direct pour la métrique analytique.

Samplers/LocalSampler

Échantillonneur par recherche locale dans un maillage de fond pour la métrique analytique.

Samplers/ComputedLocSampler

Échantillonneur par recherche locale dans un maillage de fond pour la métrique calculée par un estimateur de l'erreur *a posteriori*.

Samplers/TextureSampler

Échantillonneur par interpolation linéaire dans une texture pour la métrique analytique.

Samplers/ComputedTexSampler

Échantillonneur par interpolation linéaire dans une texture pour la métrique calculée par un estimateur de l'erreur *a posteriori*.

Samplers/KdTreeSampler (OBSOLÈTE)

Échantillonneur par kD-Tree pour la métrique analytique.

Mesures dans la métrique

Measurers/MetricFreeMeasurer

Mesureur indépendant de la métrique pour le calcul de longueur, d'air et de volume. Travaille uniquement avec l'échantillonneur uniforme.

Measurers/MetricWiseMeasurer

Mesureur dépendant de la métrique pour le calcul de longueur, d'air et de volume.

Évaluation de la qualité

Evaluators/AbstractEvaluator

Classe de base pour l'évaluation de la qualité. Elle contient les fonctions pour mesurer la qualité des voisinages élémentaires et la qualité de maillages complets.

Evaluators/MeanRatioEvaluator

Évaluation de la qualité par la mesurer du rapport des moyennes.

Evaluators/MetricConformityEvaluator

Évaluation de la qualité par la conformité à la métrique.

Déplacement de nœuds

Smoothers/VertexWise/AbstractVertexWiseSmoothers

Classe de base pour les algorithmes de déplacement de nœuds qui travaille en parallèle sur les nœuds du maillage. Elle contient les quatre pilotes pour l'exécution des implémentations séquentiel, parallèle, GLSL et CUDA.

Smoothers/VertexWise/SpringLaplaceSmoother

Déplacement de nœuds par lissage laplacien dans sa forme classique.

Smoothers/VertexWise/QualityLaplaceSmoother

Déplacement de nœuds par lissage laplacien avec de tests de qualité.

Smoothers/VertexWise/SpawnSearchSmoother

Déplacement de nœuds par force brute.

Smoothers/VertexWise/NelderMeadSmoother

Déplacement de nœuds par l'algorithme de Nelder-Mead.

Smoothers/VertexWise/MultiElemNMSmoother

Spécialisation GPU du déplacement de nœuds par Nelder-Mead utilisant l'espace de parallélisation $\{Noeuds \times Éléments\}$ (multiélément).

Smoothers/VertexWise/GradientDescentSmoother

Déplacement de nœuds par descente du gradient.

Smoothers/VertexWise/MultiElemGradDsntSmoother

Spécialisation GPU du déplacement de nœuds par descente du gradient utilisant l'espace de parallélisation $\{Noeuds \times Éléments\}$ (multiélément).

Smoothers/VertexWise/MultiPosGradDsntSmoother

Spécialisation GPU du déplacement de nœuds par descente du gradient utilisant l'espace de parallélisation $\{Noeuds \times Positions\}$ (multiposition).

Smoothers/VertexWise/PatchGradDsntSmoother

Spécialisation GPU du déplacement de nœuds par descente du gradient utilisant l'espace de parallélisation $\{Noeuds \times Éléments \times Positions\}$ (multiaxe).

Smoothers/ElementWise/AbstractElementWiseSmoothers (OBSOLÈTE)

Classe de base pour les algorithmes de déplacement de nœuds qui travaille en parallèle sur les éléments du maillage. Elle contient les quatre pilotes pour l'exécution des implémentations séquentiel, parallèle, GLSL et CUDA.

Smoothers/ElementWise/GetmeSmoother (OBSOLÈTE)

Déplacement de nœuds par la *Geometri Element Transformation Methods* (GETMe).

Modifications topologiques

Topologists/BatrTopologist

Ensemble des opérations topologiques utilisées pour préparer les cas de test.

Génération de maillage

Meshers/CpuDelaunayMesher

Génère des maillages par triangulation Delaunay pour des domaines cubiques, sphériques et en forme de coquille.

Meshers/CpuParamétriqueMesher

Génère les maillages d'un tuyau en U et d'une bouteille.

Meshers/DebugMesher

Génère divers maillages pour le débogage de l'application. Génère également le maillage structuré d'un cube ainsi qu'un cube divisé en cinq tétraèdres. Ces deux maillages sont utilisés par certains cas de test présentés dans l'analyse des résultats.

Entrées/Sorties

Serialization/JsonSerializer

Écriture sur le disque sous le format JSON des maillages produits à l'aide du logiciel

d'adaptation et de sa frontière.

Serialization/JsonDeserializer

Lecture des maillages, et de sa frontière, préalablement écrits sur le disque à l'aide de la classe *JsonSerializer*.

Serialization/PieDeserializer

Lecture des maillages et des fonctions métrique au format Pirate. La géométrie de la frontière ne peut être lue pour le moment.

ANNEXE B Pilotes des algorithmes

Séquentiel

```
void AbstractVertexWiseSmoother::smoothMeshSerial(
        Mesh& mesh,
        const MeshCrew& crew)
{
    bool isTopoEnabled =
        _schedule.topoOperationEnabled &&
        crew.topologist().needTopologicalModifications(mesh);

    _relocPassId = INITIAL_PASS_ID;
    while(evaluateMeshQualitySerial(mesh, crew))
    {
        if(isTopoEnabled)
        {
            verboseCuda = false;
            crew.topologist().restructureMesh(mesh, crew, _schedule);
            verboseCuda = true;
        }

        while(evaluateMeshQualitySerial(mesh, crew))
        {
            smoothVertices(mesh, crew,
                mesh.nodeGroups().serialGroup());
        }

        if(isTopoEnabled)
            _relocPassId = COMPARE_PASS_ID;
        else
            break;
    }
}
```

Parallèle

```
void AbstractVertexWiseSmoother::smoothMeshThread(
        Mesh& mesh,
        const MeshCrew& crew)
{
    bool isTopoEnabled =
        _schedule.topoOperationEnabled &&
        crew.topologist().needTopologicalModifications(mesh);

    uint threadCount = thread::hardware_concurrency();
    mesh.nodeGroups().setCpuWorkerCount(threadCount);

    _relocPassId = INITIAL_PASS_ID;
    while(evaluateMeshQualityThread(mesh, crew))
    {
        if(isTopoEnabled)
        {
            verboseCuda = false;
            crew.topologist().restructureMesh(mesh, crew, _schedule);
            verboseCuda = true;
        }

        while(evaluateMeshQualityThread(mesh, crew))
        {
            std::mutex mutex;
            std::condition_variable cv;
            std::atomic<int> done( 0 );
            std::atomic<int> step( 0 );

            vector<thread> workers;
            for(uint t=0; t < threadCount; ++t)
            {
                workers.push_back(thread([&, t]() {
                    size_t groupCount = mesh.nodeGroups().count();
                    for(size_t g=0; g < groupCount; ++g)
                    {
                        smoothVertices(mesh, crew,
                            mesh.nodeGroups().parallelGroups()[g].↩
                                allDispatchedNodes[t]);

                        if(g < groupCount-1)
                        {
```

```
                            std::unique_lock<std::mutex> lk(mutex);
                            if(done.fetch_add( 1 ) == threadCount-1)
                            {
                                ++step;
                                done.store( 0 );
                                cv.notify_all();
                            }
                            else
                            {
                                cv.wait(lk, [&](){ return step > g; });
                            }
                        }
                    }
                }));
            }

            for(uint t=0; t < threadCount; ++t)
            {
                workers[t].join();
            }
        }

        if(isTopoEnabled)
            _relocPassId = COMPARE_PASS_ID;
        else
            break;
    }
}
```

GLSL

```
void AbstractVertexWiseSmoother::smoothMeshGlsl(
        Mesh& mesh,
        const MeshCrew& crew)
{
    initializeProgram(mesh, crew);

    mesh.updateGlslTopology();
    mesh.updateGlslVertices();
    crew.updateGlslData(mesh);

    _vertSmoothProgram.pushProgram();
    crew.setPluginGlslUniforms(mesh, _vertSmoothProgram);
    setVertexProgramUniforms(mesh, _vertSmoothProgram);
    _vertSmoothProgram.popProgram();

    size_t groupCount = mesh.nodeGroups().count();
    uint threadCount = thread::hardware_concurrency();
    mesh.nodeGroups().setCpuWorkerCount(threadCount);
    mesh.nodeGroups().setGpuDispatcher(glslDispatcher());

    bool isTopoEnabled =
        _schedule.topoOperationEnabled &&
        crew.topologist().needTopologicalModifications(mesh);

    _relocPassId = INITIAL_PASS_ID;
    while(evaluateMeshQualityGlsl(mesh, crew))
    {
        if(isTopoEnabled)
        {
            verboseCuda = false;
            mesh.fetchGlslVertices();
            crew.topologist().restructureMesh(mesh, crew, _schedule);
            mesh.updateGlslTopology();
            mesh.updateGlslVertices();
            verboseCuda = true;

            groupCount = mesh.nodeGroups().count();
        }

        while(evaluateMeshQualityGlsl(mesh, crew))
        {
```

```
std::mutex mutex;
std::condition_variable groupCv;
std::condition_variable memcpyCv;
std::atomic<int> moveDone( 0 );
std::atomic<int> stepDone( 0 );
std::atomic<bool> memcpyDone( false );

vector<thread> workers;
for(uint t=0; t < threadCount; ++t)
{
    workers.push_back(thread([&, t]() {
        for(size_t g=0; g < groupCount; ++g)
        {
            const NodeGroups::ParallelGroup& group =
                mesh.nodeGroups().parallelGroups()[g];

            smoothVertices(mesh, crew,
                group.cpuOnlyDispatchedNodes[t]);

            if(g < groupCount)
            {
                std::unique_lock<std::mutex> lk(mutex);
                if(moveDone.fetch_add( 1 ) == threadCount)
                {
                    ++stepDone;
                    moveDone.store( 0 );
                    memcpyDone = false;
                    groupCv.notify_all();
                }
                else
                {
                    groupCv.wait(lk, [&](){ return stepDone > g;↩
                        });
                }

                memcpyCv.wait(lk, [&](){ return memcpyDone.load↩
                    (); });
            }
        }
    }));
}

_vertSmoothProgram.pushProgram();
mesh.bindGlShaderStorageBuffers();
```

```
for(size_t g=0; g < groupCount; ++g)
{
    const NodeGroups::ParallelGroup& group =
            mesh.nodeGroups().parallelGroups()[g];

    const NodeGroups::GpuDispatch& dispatch = group.gpuDispatch;


    glBindBuffer(GL_SHADER_STORAGE_BUFFER,
                 mesh.glBuffer(EMeshBuffer::VERT));

    if(dispatch.workgroupCount.x *
       dispatch.workgroupCount.y *
       dispatch.workgroupCount.z > 0)
    {
        _vertSmoothProgram.setInt("GroupBase", dispatch.↩
            gpuBufferBase);
        _vertSmoothProgram.setInt("GroupSize", dispatch.↩
            gpuBufferSize);

        glDispatchComputeGroupSizeARB(
            dispatch.workgroupCount.x,
            dispatch.workgroupCount.y,
            dispatch.workgroupCount.z,
            dispatch.workgroupSize.x,
            dispatch.workgroupSize.y,
            dispatch.workgroupSize.z);

        glMemoryBarrier(GL_ALL_BARRIER_BITS);


        // Fetch subsurface vertex positions from GPU
        size_t subsurfaceSize =
                group.subsurfaceRange.end -
                group.subsurfaceRange.begin;

        if(subsurfaceSize > 0)
        {
            subsurfaceSize *= sizeof(GpuVert);
            size_t subsurfaceBase = group.subsurfaceRange.begin ↩
                * sizeof(GpuVert);
            GpuVert* boundVerts = static_cast<GpuVert*>(
                glMapBufferRange(GL_SHADER_STORAGE_BUFFER,
                    subsurfaceBase, subsurfaceSize,
```

```
                GL_MAP_READ_BIT));

        for(size_t vId = group.subsurfaceRange.begin, bId=0;
            vId < group.subsurfaceRange.end; ++vId, ++bId)
        {
            MeshVert vert(boundVerts[bId]);
            mesh.verts[vId] = vert;
        }

        glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
    }
}


// Synchronize with CPU workers
if(g < groupCount)
{
    std::unique_lock<std::mutex> lk(mutex);
    if(moveDone.fetch_add( 1 ) == threadCount)
    {
        ++stepDone;
        moveDone.store( 0 );
        memcpyDone = false;
        groupCv.notify_all();
    }
    else
    {
        groupCv.wait(lk, [&](){ return stepDone > g; });
    }
}


// Send boundary vertex positions to GPU
size_t boundarySize =
        group.boundaryRange.end -
        group.boundaryRange.begin;

if(boundarySize > 0)
{
    boundarySize *= sizeof(GpuVert);
    size_t boundaryBase = group.boundaryRange.begin * sizeof↩
        (GpuVert);
    GpuVert* boundVerts = static_cast<GpuVert*>(
        glMapBufferRange(GL_SHADER_STORAGE_BUFFER,
```

```
                        boundaryBase, boundarySize,
                        GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_RANGE_BIT))↩
                            ;

                    for(size_t vId = group.boundaryRange.begin, bId=0;
                        vId < group.boundaryRange.end; ++vId, ++bId)
                    {
                        GpuVert vert(mesh.verts[vId]);
                        boundVerts[bId] = vert;
                    }

                    glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
                }

                glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
                glMemoryBarrier(GL_ALL_BARRIER_BITS);


                // Wake-up threads
                mutex.lock();
                memcpyDone = true;
                mutex.unlock();
                memcpyCv.notify_all();
            }
            _vertSmoothProgram.popProgram();

            for(uint t=0; t < threadCount; ++t)
            {
                workers[t].join();
            }
        }

        if(isTopoEnabled)
            _relocPassId = COMPARE_PASS_ID;
        else
            break;
    }

    // Fetch new vertex positions
    mesh.fetchGlslVertices();
    mesh.clearGlslMemory();
    crew.clearGlslMemory(mesh);
}
```

CUDA

```
void AbstractVertexWiseSmoother::smoothMeshCuda(
        Mesh& mesh,
        const MeshCrew& crew)
{
    _installCudaSmoother();

    mesh.updateCudaTopology();
    mesh.updateCudaVertices();
    crew.updateCudaData(mesh);

    crew.setPluginCudaUniforms(mesh);

    size_t groupCount = mesh.nodeGroups().count();
    uint threadCount = thread::hardware_concurrency();
    mesh.nodeGroups().setCpuWorkerCount(threadCount);
    mesh.nodeGroups().setGpuDispatcher(cudaDispatcher());

    bool isTopoEnabled =
        _schedule.topoOperationEnabled &&
        crew.topologist().needTopologicalModifications(mesh);

    _relocPassId = INITIAL_PASS_ID;
    while(evaluateMeshQualityCuda(mesh, crew))
    {
        if(isTopoEnabled)
        {
            verboseCuda = false;
            mesh.fetchCudaVertices();
            crew.topologist().restructureMesh(mesh, crew, _schedule);
            mesh.updateCudaTopology();
            mesh.updateCudaVertices();
            verboseCuda = true;

            groupCount = mesh.nodeGroups().count();
        }

        while(evaluateMeshQualityCuda(mesh, crew))
        {
            std::mutex mutex;
            std::condition_variable groupCv;
            std::condition_variable memcpyCv;
```

```
std::atomic<int> moveDone( 0 );
std::atomic<int> stepDone( 0 );
std::atomic<bool> memcpyDone( false );

vector<thread> workers;
for(uint t=0; t < threadCount; ++t)
{
    workers.push_back(thread([&, t]() {
        for(size_t g=0; g < groupCount; ++g)
        {
            const NodeGroups::ParallelGroup& group =
                mesh.nodeGroups().parallelGroups()[g];

            smoothVertices(mesh, crew,
                group.cpuOnlyDispatchedNodes[t]);

            if(g < groupCount)
            {
                std::unique_lock<std::mutex> lk(mutex);
                if(moveDone.fetch_add( 1 ) == threadCount)
                {
                    ++stepDone;
                    moveDone.store( 0 );
                    memcpyDone = false;
                    groupCv.notify_all();
                }
                else
                {
                    groupCv.wait(lk, [&](){ return stepDone > g;↩
                        });
                }

                memcpyCv.wait(lk, [&](){ return memcpyDone.load↩
                    (); });
            }
        }
    }));
}


for(size_t g=0; g < groupCount; ++g)
{
    const NodeGroups::ParallelGroup& group =
            mesh.nodeGroups().parallelGroups()[g];
```

```
    const NodeGroups::GpuDispatch& dispatch = group.gpuDispatch;

    if(dispatch.workgroupCount.x *
       dispatch.workgroupCount.y *
       dispatch.workgroupCount.z > 0)
    {
        _launchCudaKernel(dispatch);

        // Fetch subsurface vertex positions from GPU
        fetchCudaSubsurfaceVertices(mesh.verts, group);
    }

    // Synchronize with CPU workers
    if(g < groupCount)
    {
        std::unique_lock<std::mutex> lk(mutex);
        if(moveDone.fetch_add( 1 ) == threadCount)
        {
            ++stepDone;
            moveDone.store( 0 );
            memcpyDone = false;
            groupCv.notify_all();
        }
        else
        {
            groupCv.wait(lk, [&](){ return stepDone > g; });
        }
    }

    // Send boundary vertex positions to GPU
    sendCudaBoundaryVertices(mesh.verts, group);


    // Wake-up threads
    mutex.lock();
    memcpyDone = true;
    mutex.unlock();
    memcpyCv.notify_all();
}


for(uint t=0; t < threadCount; ++t)
{
```

```
                workers[t].join();
            }
        }

        if(isTopoEnabled)
            _relocPassId = COMPARE_PASS_ID;
        else
            break;
    }


    // Fetch new vertex positions
    mesh.fetchCudaVertices();
    mesh.clearCudaMemory();
    crew.clearCudaMemory(mesh);
}
```

ANNEXE C Tableaux des histogrammes

Histogramme des techniques d'échantillonnage

Tableau C.1 Qualités des éléments pour les maillages lissés avec un ratio d'aspect $A = 16$

Qualités	Initial	Analytique	Rech. loc.	Texture
0.00 - 0.05	1	0	0	0
0.05 - 0.10	141	0	0	0
0.10 - 0.15	2651	77	82	69
0.15 - 0.20	14051	1612	1559	1369
0.20 - 0.25	34978	9360	9262	9168
0.25 - 0.30	60159	26100	26168	27040
0.30 - 0.35	76780	47084	47185	49507
0.35 - 0.40	81974	62337	62509	66297
0.40 - 0.45	82447	69150	69585	74902
0.45 - 0.50	78947	71285	71482	77310
0.50 - 0.55	75304	72847	72978	77134
0.55 - 0.60	70506	76164	76131	77034
0.60 - 0.65	62706	81301	81373	78136
0.65 - 0.70	53228	81381	81348	76024
0.70 - 0.75	38177	71156	71115	65994
0.75 - 0.80	25956	54286	53886	49149
0.80 - 0.85	14777	34345	34125	31397
0.85 - 0.90	7395	17765	17523	16173
0.90 - 0.95	2659	6203	6156	5766
0.95 - 1.00	364	748	734	732

Histogramme des différences d'implémentation

Tableau C.2 Qualités des maillages lissés séquentiellement et parallèlement sur CPU et GPU

Qualités	Initial	Séquentiel	Parallèle	GLSL	CUDA
0.00 - 0.05	366	0	0	0	0
0.05 - 0.10	5862	455	474	470	483
0.10 - 0.15	5084	3695	3746	3764	3761
0.15 - 0.20	4024	4109	4074	4076	4047
0.20 - 0.25	4102	4558	4521	4505	4520
0.25 - 0.30	4713	5521	5539	5520	5512
0.30 - 0.35	6275	7162	7130	7150	7152
0.35 - 0.40	8254	8496	8492	8508	8490
0.40 - 0.45	8018	8927	8902	8899	8916
0.45 - 0.50	6526	7553	7564	7556	7572
0.50 - 0.55	4549	5510	5521	5515	5519
0.55 - 0.60	2810	3603	3630	3635	3622
0.60 - 0.65	1642	2167	2168	2163	2171
0.65 - 0.70	843	1137	1129	1131	1123
0.70 - 0.75	453	575	572	572	575
0.75 - 0.80	203	254	261	260	256
0.80 - 0.85	108	83	83	82	87
0.85 - 0.90	18	40	38	39	37
0.90 - 0.95	6	12	13	12	14
0.95 - 1.00	1	0	0	0	0